MODELING OF ANDROID SOFTWARE BEHAVIOR FEATURE AND ITS APPLICATIONS IN MALICIOUS PROGRAM ANALYSIS

MING FAN

PhD

The Hong Kong Polytechnic University

This programme is jointly offered by The Hong Kong Polytechnic University and Xi’an Jiaotong University

2019

The Hong Kong Polytechnic University
Department of Computing
Xi’an Jiaotong University
Department of Computer Science and Technology

Modeling of Android Software Behavior Feature and Its Applications in Malicious Program Analysis

Ming Fan

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

March 2019

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

____________________ (Signed)

Ming Fan (Name of student)

Abstract

Over the past ten years, due to three main advantages (e.g., the openness of source code, the richness of hardware selection, and millions of applications (apps)), Android has become the most popular mobile operating system. Meanwhile, it has also become the major target of mobile malware. The rapid increase in the number of Android malware poses great threats to the smartphone users, such as financial charge, information collection, and remote control. Thus, the in-depth study of the security issues of mobile apps is of great significance to the sound development of the smartphone ecosystem. However, existing malware analysis approaches are facing three main challenges, including morphological diversity of malicious code, lack of labeled dataset, and labor-intensive manual feature engineering process. Therefore, it is important to propose effective and efficient malware analysis approaches. To further study two sub problems in mobile security, i.e., malware detection and familial identification, two kinds of behavior models and four different types of features are proposed from the novel perspective of feature engineering. In this thesis, malware detection aim to detect whether a given app is malicious or not and familial identification aim to classify the malware samples to their corresponding families.

Firstly, to overcome the low accuracy and efficiency problems caused by the morphological diversity of malicious code, the sensitive subgraph is first constructed as our analysis model. It can not only depict the sensitive behavior but also can be resilient to obfuscation techniques. Based on the sensitive subgraph, for malware detection, a structure-based feature called maximum sensitive subgraph is proposed to depict the most sensitive behavior of a given app. Based on the proposed feature, this study designs and implements DAPASA, a approach that detects Android piggybacked apps. DAPASA can not only detect the piggybacked apps dependently but also has the ability to complement permission- and API-based approaches from a new perspective of the invocation structure. For familial identification, a new feature called frequent sensitive subgraphs (fregraphs) is proposed to represent the common behaviors of malware samples that belong to the same family. Then, this study designs and implements FalDroid, an approach that automatically classifies Android malware into their corresponding families and selects representative malware samples in each family accordance with fregraphs. In this way, FalDroid can effectively reduce the analytical workload and accelerate malware analysis.

Then, to overcome the limitation of existing supervised learning approaches in handling unlabeled dataset, the graph structure of sensitive subgraph is abstracted by leveraging the graph embedding techniques and a new feature called SRA is proposed to depict the similarity relationships of structural roles of sensitive API call nodes in a graph. The SRA feature can not only retain the semantic information of the graph but also can transform the high-cost graph matching into an easy-to-compute similarity calculation between vectors. Then this study designs and implements GefDroid, an approach that constructs a malware link network to depict the similarity relationships between all samples based on the SRA feature. In this way, this study can handle the unlabeled samples with unsupervised learning.

After that, to ease the labor-intensive manual feature engineering process, this study proposes techniques that summarize the existing knowledge contained in magnanimity information of natural language documents and generates a novel type of features called sensitive behavior, which is represented as verb-objective phrases that are easy to understand. This study designs and implements CTDroid, an automatic feature engineering system. By using CTDroid, a set of informative features is constructed from technical blogs that can be utilized for Android malware analysis.

The four approaches are evaluated on the datasets that consist of real benign apps and malware samples. The results of extensive experiments demonstrate that: DAPASA achieves good performance on detecting piggybacked apps with a true positive rate of 95% and a false positive rate of 0.7%; FalDroid can correctly classify 94.2% of malware samples into their families using approximately 4.6 seconds per app; GefDroid can achieve high agreements (0.707-0.883 in term of NMI) between our clustering results and the ground truth datasets; The features extracted by CTDroid perform well for malware analysis and are more informative than those of state-of-the-art approaches.

Keywords: Android Application; Behavior Modeling; Malware Detection; Familial Identification; Sensitive Subgraph

Publications

1. **Ming Fan**, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu, "Graph Embedding based Familial Analysis of Android Malware using Unsupervised Learning", in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*, Accepted to appear, 2019.

2. **Ming Fan**, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu, "Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis", IEEE Transaction on Information Forensics and Security (IEEE TIFS), August, 2018.

3. **Ming Fan**, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu, "DAPASA: Detecting Android Piggybacked Apps Through Sensitive Subgraph Analysis", IEEE Transaction on Information Forensics and Security (IEEE TIFS), August, 2017.

4. **Ming Fan**, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu, "Frequent Subgraph based Familial Classification of Android Malware". in *Proceedings of IEEE 27th International Symposium on Software Reliability Engineering(ISSRE'16)*, Ottawa, Canada, October 23-27, 2016. (**Best Research Paper Award**)

5. Zhenzhou Tian, Ting Liu, Qinghua Zheng, Eryue Zhuang, **Ming Fan**, and Zijiang

Yang, “Reviving Sequential Program Birthmarking for Multithreaded Software Plagiarism Detection”, IEEE Transaction on Software Engineering (IEEE TSE), May, 2018.

6. Zhenzhou Tian, Ting Liu, Qinghua Zheng, **Ming Fan**, Eryue Zhuang, and Zijiang Yang, “Exploiting Thread-related System Calls for Plagiarism Detection of Multithreaded Programs”, The Journal of Systems and Software (JSS), Sep, 2016.

7. Zhenzhou Tian, Qinghua Zheng, Ting Liu, **Ming Fan**, Eryue Zhuang, and Zijiang Yang, “Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences”, IEEE Transaction on Software Engineering (IEEE TSE), July, 2015.

8. Zhenzhou Tian, Qinghua Zheng, Ting Liu, **Ming Fan**, Xiaodong Zhang, and Zijiang Yang, “Plagiarism Detection for Multithreaded Software Based on Thread-Aware Software Birthmarks”, in *Proceedings of ACM 22nd International Conference on Program Comprehension (ICPC)*, Hyderabad, India, June 2-3, 2014.

9. Zhenzhou Tian, Qinghua Zheng, **Ming Fan**, Eryue Zhuang, Haijun Wang, Ting Liu, “DBPD: A Dynamic Birthmark-based Software Plagiarism Detection Tool”, in *Proceedings of 6th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Hyatt Regency, Vancouver, Canada, July 1-3, 2014. (**Best Demo Award**)

Acknowledgements

First and foremost, I am deeply grateful to all the nice and great people I have met or known. Although many of them may not be mentioned, what I learned from them inspires everything I have accomplished.

I want to extend my heartfelt gratitude to my supervisor, Assistant Prof. Xiapu Luo, whose patient guidance, valuable suggestions and constant encouragement make me successfully complete this thesis. His conscientious academic spirit and modest, open-minded personality inspire me both in academic study and daily life. He gives me much help and advice during the whole process of my writing, which has made my accomplishments possible.

Also, I would like to express my sincere gratitude to all the professors who have taught me in this university. Their instructions have helped broaden my horizon and their enlightening teaching has provided me with a solid foundation to accomplish this paper and will always be of great value for my future career and academic research.

There are also many other teachers and classmates that helped me a lot during my study period. I would like to thank Prof. Jun Liu, Prof. Ting Liu, Lei Xue, Le Yu, Zhou Xu, Hao Zhou, Jiachi Chen, Ting Chen, Chenxu Wang and Tao Zhang for providing novel inspirations and suggestions when I discussed with them, and their rigorous scientific approaches helped me become a well-trained researcher. I would also like to thank all the members of my group. The friendly working environment created by them is essential for my study, and the time I worked and discussed with them is a nice memory in my life.

Last but not least, I would like to express my special thanks to my parents, whose care and support motivate me to move on and make me want to be a better person.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android is a Linux-based free and open source operating system, which is led and developed by Google and the Open Handset Alliance in 2007. It is widely used in mobile devices such as smartphones and tablets. The first Android smartphone was released in October 2008, and Android gradually expanded to tablets and other areas such as TVs, digital cameras, smart watches, game consoles, etc. Compared to traditional Symbian operating system, Android has the following three advantages. 1) Open source: The open source code, free development software, and communities have attracted more and more developers. The huge competition makes Android more and more mature in an open platform. 2) Rich hardware selection: In order to attract more users, many manufacturers transform the Android system and produce various products with different features without affecting data synchronization, software compatibility and other functions. 3) Millions of applications (apps) to download: There are many Android app markets that can provide users with a variety of rich app downloads, such as official Google Play [20], and third-party market Anzhi [14]. These advantages make Android become the most popular mobile operating system. According to data provided by research firm Gartner, in the first quarter of 2018, Android has occupied 85.9% of the market share of mobile operating system [55].

Meanwhile, Android has become the major target of 97% of mobile malware [81]. A recent security report shows that on average, 38,000 new mobile malware samples were captured per day during the third quarter of 2016 [78]. The rapid growth of malware has caused tremendous harm to mobile smartphone users, including financial charge, information collection, and remote control. Thus, the in-depth study of the security issues of mobile applications is of great significance to the sound development of the smart phone ecosystem.

This thesis focuses on two sub-problems of malware detection and familial identification in mobile application security issues. The malware detection problem is defined as follows. Given a mobile app, how do we effectively detect it is malicious through program analysis, which is a binary classification problem. The familial identification problem is defined as follows. Given a malicious app, how do we effectively classify it into its corresponding family. It is a a multi-classification problem. Here a malware family means a group of malware samples that conduct similar malicious behaviors.

For these two major research problems, the existing malware analysis approaches are facing the following three main challenges:

- **Morphological diversity of malicious code:** To bypass existing malware detection approaches, malicious code creators create many different types of variants by constantly modifying the malicious source code. In addition, the increasingly sophisticated obfuscation technologies would change the structure of malicious code, making the naive approaches such as signature-based approaches more and more difficult to be detected.
- **Lack of labeled dataset:** Since the malware samples generally contain thousands lines of code, it is time-consuming and labor-intensive to manually read the code and label a large scale of malware samples with family names. Moreover, since

classifiers are trained using known malware samples, they cannot correctly classify new malware samples from unknown families. Note that retraining the classifier model for every new sample may be impractical.

- **Labor-intensive manual feature engineering process:** The effectiveness of the above approaches primarily depends on the manual feature engineering process, which is time-consuming and labor-intensive based on human knowledge and intuition. Specifically, to perform malware analysis with high performance, the researchers need to manually inspect the malicious activities of malware samples and summarize the hypotheses about common behaviors that malware share but benign apps do not. Furthermore, the summarized hypotheses might vary from different inspected malware samples, thus constructing different feature spaces for different datasets.

The overview architecture of the thesis is illustrated in Fig. 1.1. To overcome the above three challenges, two different types of software behavior model are proposed. They are graph-based model and text-based model. Then, for the graph-based model, the thesis proposes the maximum sensitive subgraph feature and the frequent sensitive subgraph feature to solve the problems brought by the first challenge in malware detection and familial identification. After that, to overcome the second challenge, the graph structure of sensitive subgraph is abstracted by leveraging the graph embedding techniques, and a new graph embedding feature called SRA is proposed to perform familial clustering. Finally, to ease the labor-intensive manual feature engineering process, the thesis summarizes the existing knowledge contained in magnanimity information of natural language documents and generates a novel type of features called sensitive behavior to perform malware analysis. The details of the four research contents are introduced below:

1) The construction of maximum sensitive subgraph feature and its application in

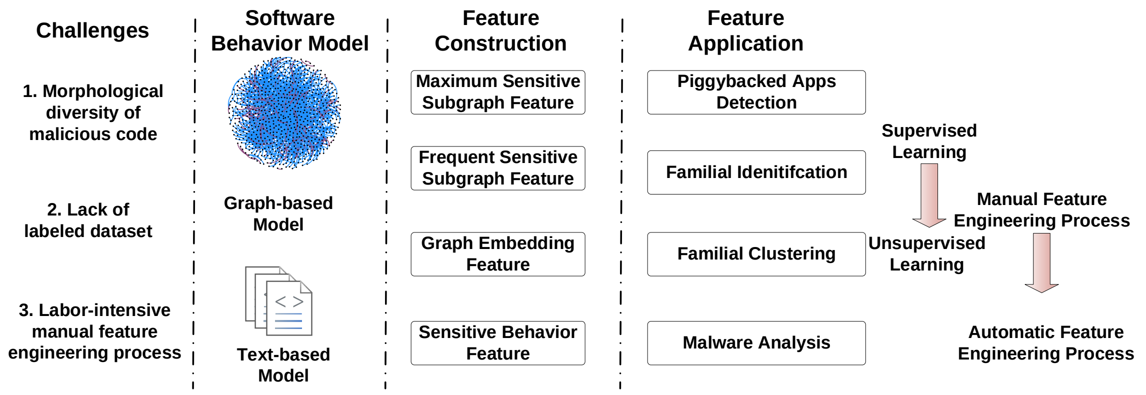

Challenges
1. Morphological diversity of malicious code
2. Lack of labeled dataset
3. Labor-intensive manual feature engineering process
Software Behavior Model
Graph-based Model
Text-based Model
Feature Construction
Maximum Sensitive Subgraph Feature
Frequent Sensitive Subgraph Feature
Graph Embedding Feature
Sensitive Behavior Feature
Feature Application
Piggybacked Apps Detection
Familial Idenitifcation
Familial Clustering
Malware Analysis
Supervised Learning
Unsupervised Learning
Manual Feature Engineering Process
Automatic Feature Engineering Process

Figure 1.1: Overview architecture of the thesis

piggybacked apps detection

According to the results provided by existing work, piggybacking is one of the most common techniques utilized by malware authors to piggyback malicious payloads on popular apps to produce malware. About 86% of malware samples were piggybacked versions of legitimate apps with malicious payloads [155]. Piggybacked apps pose significant security threats, and effective techniques to detect them are necessary. Piggybacked apps are constantly modified which makes them difficult to detect. However, based on an empirical analysis of piggybacked apps, we find that to perform its malicious task, the injected malicious components invoke more sensitive API calls than the benign app does. Moreover, in the injected malicious components, the cohesion of sensitive API calls, which is measured by calling distances, is higher than that in the benign app.

Based on these two observations, we start with the construction of a static function-call graph (FCG) of a given app. Then to differentiate the maliciousness of different sensitive API calls, we calculate the sensitivity coefficients for each sensitive call API through a term frequency-inverse document frequency (TF-IDF)-like measure. After that, we divide the FCG into a set of subgraphs and selects the subgraph with the highest sensitivity coefficient as the maximum sensitive subgraph to profile the most suspicious behavior of the given app. Finally, five features are constructed from the maximum sensitive subgraph and fed into machine learning algorithms to detect whether the app is piggybacked or benign.

Based on the maximum sensitive subgraph feature, we develop DAPASA, a novel system for the detection of Android piggybacked apps. Extensive experimental results demonstrate that DAPASA can achieve good performance with a true positive rate of 95% and a false positive rate of 0.7%. In addition, DAPASA can complement the permission- and API-based approaches from a new perspective of the invocation structure.

2) The construction of frequent sensitive subgraph feature and its application in familial identification

The analysis of each malware sample requires ample time. Hence, the sheer number of malware samples overwhelms malware analysis systems. The majority of new malware samples are polymorphic variants of known malware. Thus, to accelerate malware analysis, we can classify malware samples into various families and then select representative samples from each family. However, polymorphic variants of Android malware that belong to the same family perform the same malicious activities with different implementations. To address this limitation, we first construct sensitive API call related graph (SARG) through graph analysis techniques on FCG. Then, SARG is initially divided into a set of subgraphs using community detection algorithms. With subgraph matching and clustering techniques, the sensitive subgraphs used by most samples in one family are defined as the frequent sensitive subgraphs (fregraphs). After that, each fregraph is assigned with a weight score and a feature vector is constructed for each app. Finally, known machine learning algorithms can be applied to perform the familial identification task.

Based on the fregraph feature, we develop FalDroid, an automatic system for identifying

Android malware and selecting representative samples of each family. Extensive experimental results demonstrate that FalDroid achieves a 94.2% accuracy and only requires approximately 4.6 sec to process an app.

3) The construction of graph embedding feature and its application in familial clustering

The above two approaches focus on the classifying of unlabeled malware samples using supervised learning methods. However, it is time-consuming and labor-intensive to label a large scale of malware samples with family names. Moreover, since classifiers are trained using known malware samples, they cannot correctly classify new malware samples from unknown families. To solve the second challenge, we propose a novel feature called SRA to depict the similarity relationships of structural roles of sensitive API call nodes in a graph. The SRA feature can not only retain the semantic information of the graph but also can transform the high-cost graph matching into an easy-to-compute similarity calculation between vectors. In particular, we construct a malware link network (MLN) to represent the similarity relationships among samples based on their similar SRAs. Finally, we apply community detection algorithms to group the samples into a set of clusters.

Based on the SRA feature, we propose and develop GefDroid, a novel system for familial analysis of Android malware by using unsupervised learning. Extensive experimental results demonstrate that GefDroid can achieve high agreements (0.707-0.883 in term of NMI) between our clustering results and the ground truth datasets. Furthermore, GefDroid requires only linear run-time overhead and takes around 8.6s to analyze a sample on average, which is considerably faster than the prior arts.

4) The construction of sensitive behavior feature and its application in malware analysis

The effectiveness of the above approaches primarily depends on the manual feature engineering process, which is time-consuming and labor-intensive based on human knowledge and intuition. To solve the third challenge, we aim to automatically engineer informative features from existing knowledge learned by experts and apply them on malware analysis. Given that it is hard to automatically recognize the harmful activities in the magnanimity information of thousands of technical blogs, we first leverage natural language processing (NLP) techniques to parse the contents in blogs into a uniform structure. Then, we propose a clustering-based approach to extract frequent behaviors that have close relations with Android system and regard them as sensitive behaviors. However, there also exists a semantic gap between the sensitive behaviors and the programming language. Thus, we propose two semantic matching rules to bridge the gap between the sensitive behavior and the programming language based on the analysis of descriptions of Android concrete features, as well as the keywords in the app code.

Based on the sensitive behavior feature, we propose and develop CTDroid, an automatic feature engineering system used for malware analysis. Experimental results show that our proposed features can perform well for malware detection and familial identification. Furthermore, these features are more informative than those of state-of-the-art approaches.

The rest of this thesis is organized as follows. Chapter 2 introduces the related work. Chapter 3 proposes a feature called maximum sensitive subgraph feature, and implements DAPASA, a novel approach for the detection of Android piggybacked apps. Chapter 4 proposes a feature called frequent sensitive subgraph feature, and implements FalDroid, a novel approach for identifying Android malware and selecting representative samples of each family. Chapter 5 proposes a feature called SRA, and implements GefDroid, a novel approach for familial analysis of Android malware by using unsupervised learning. Chapter 6 proposes a feature called sensitive behavior feature, and implements CTDroid, an automatic feature engineering system used for malware analysis. Finally, Chapter 7 concludes this thesis and discusses future work.

Chapter 2

Literature Review

The rapid increase in the number of Android malware poses great threats to the smartphone users, such as financial charge, information collection, and remote control. The development of malware analysis technology has quickly become a hot topic in the field of software security, and has achieved a lot of excellent research results. In this chapter, we first introduce the widely-used datasets and then discuss three main types of work that are related to this thesis, including malware analysis, graph embedding techniques and NLP techniques for Android.

2.1 Malware Datasets

The malware analysis is inseparable from the support of effective datasets. There are three widely-used datasets: Genome Project dataset [156], Drebin dataset [34] and the DroidBench dataset [3].

The Genome Project dataset is constructed by Zhou and Jiang with a duration over more than one year. The dataset contains 1,260 malware samples in 49 families. The team spends a lot of manpower and resources to manually analyze the samples in the dataset.

The results demonstrate that: 1) More than 86% malware samples are repackaged versions of legitimate apps with malicious payloads, which suggests that policing repackaged apps is necessary of the Android markets. 2) Around one third (36.7%) of the collected malware samples leverage root-level exploits to fully compromise the Android security, posing the highest level of threats to users' security and privacy. 3) 93.0% malware samples turn the infected phones into bots for remote control. Specifically, they use the HTTP-based web traffic to receive bot commands from their C&C servers.

To foster research on Android malware and to enable a comparison of different detection approaches, Drebin dataset extends the Genome project dataset and is built by by Arp *et al.* The Drebin dataset contains 5,560 malware samples in 179 families. The samples are collected in the period of August 2010 to October 2012.

The DroidBench dataset is constructed by Fritz *et al.* It is an open test suite for evaluating the effectiveness of taint-analysis tools specifically for Android apps. The suite can be used to assess both static and dynamic taint analyses, but in particular it contains test cases for interesting static-analysis problems (object sensitivity, field sensitivity, tradeoffs in access-path lengths etc.) as well as for Android-specific challenges like correctly modeling an app's lifecycle, adequately handling asynchronous callbacks and interacting with the UI.

2.2 Malware Analysis

Existing malware analysis studies fall into two general categories: 1) signature-based approaches and 2) machine learning-based approaches.

2.2.1 Signature-based Approaches

Signature-based approaches look for specific patterns of malware behavior. The earliest signature is the digital signature of each Android app. According to the Android system requirements, each app must have a valid digital signature to be uploaded to the app market. Each digital signature uniquely identifies the app, so the most primitive malware detection approach is to check whether the digital signature of an app is in a database of malicious signature. However, the app code can be modified and re-signed by repackaging, and so this approach is very easy to bypass.

Enck *et al.* [56] proposed the Kirin security service for Android, which designs nine rule templates to match the undesirable properties in security configuration bundled with apps. For example, if one app requests the permissions SEND_SMS and WRITE_SMS, it will be regarded as a malicious app. Grace *et al.* [70] proposed a proactive scheme to spot zero-day Android malware, and developed a system called RiskRanker to analyze whether an app exhibits malicious behavior. Zhou *et al.* [148] proposed a permission-based behavioral footprinting scheme to detect new samples of known malware families and then applied a heuristic-based filtering scheme to identify inherent behaviors of unknown malware families.

Unlike signatures that have no program semantics, Feng *et al.* [61] proposed a semantic-based malware detection approach named as Apposcopy. Apposcopy proposes a high-level specification language for describing semantic characteristics of malware families and a powerful static analysis for checking whether a given app matches the existing signatures. For example, a signature *flow(s, DeviceId, s, Intent)* extracted from the *golddream* family indicates that the samples in the family contain the data flow which collects the device ID information and sends it to the network. However, Apposcopy relies on the security experts to carefully analyze the samples in each family and summarizes the signatures of malicious behaviors, which requires a lot of manpower and resources. Moreover, the family signatures constructed by this approach are only valid for known families and cannot detect unknown family samples. Thus, Feng *et. al* [62] proposed Astroid, which automatically generates the malware signature by analyzing a maximally suspicious common subgraph that is shared between all known instances of a malware family. The subgraph is used to depict the common malicious behavior of malware samples that belong to the same family.

2.2.2 Machine Learning-based Approaches

Machine learning-based approaches extract different features from app code and apply standard machine learning algorithms to perform a classification task. According to the types of extracted features, these approaches can be categorized into four groups.

1) Permission, API, and Components

Existing works [137, 100, 37, 91, 85] analyze the *AndroidManifest.xml* file and extract the requested permissions. Each permission is regarded as a feature. Au *et al.* [36] proposed PScout, which statically analyzes the Android system source code, and extracts the mapping relationship between API calls and their required permissions. The API call is a more granular feature than permission. Aafer *et al.*[31] proposed DroidAPIMiner, which extracts five different types of API calls as features, including app-specific resources API calls, framework resources API calls, DVM related resources API calls, system resources API calls, and utilities API calls. Unlike these static API features, other works [152, 79] use the system call APIs that are executed by the dynamic execution program as features.

Except for the typical permission- and API-based features, several works [34, 141, 92, 121, 138, 129] find that the filter intents, strings, and Android components can also be used as effective features to distinguish the malware samples and benign apps. Arp *et al.* [34] proposed Drebin, which uses the intents (e.g., *BOOT_COMPLETED* that is used to trigger malicious activity directly after rebooting the smartphone), hardware components (e.g., GPS and network modules), Android components (e.g., *activities, services, content providers, broadcast receivers*), and network address (e.g., hostnames and URLs), as features. Based on the above extracted features, Garcia *et al.* [66] proposed RevealDroid, which adds the occurred frequency of reflection methods and the system calls used in native code as new features, thus improving the resilience ability to obfuscation techniques.

2) Dalvik Code

The dalvik code obtained by decompiling the APK file is rich in semantics and contains classes, methods, and instructions of the programs. Zhou *et al.* [155] proposed DroidMOSS, which directly uses the opcodes and the operands as features. Canfora and Hanna [44, 74] transformed the opcode feature by leveraging the k-gram techniques [133]. The sliding window with step k is used to divide the opcode sequence into features consisting of k opcodes, thus improving the robustness of the feature. Suarez *et al.* [127] proposed Dendroid, which uses the basic block of app code as features. The basic block refers to the sequence of statements executed sequentially by the program. There is only one entry and one exit. The entry is the first statement, and the exit is the last statement. For basic blocks, execution only enters from its entry, exits from the exit, and does not contain any jump statements.

3) Metedata

Metadata refers to additional descriptive information that is not related to the app's own code, such as the app's downloads, functionality descriptions, category information, and others. This kind of information can improve the existing features from another new perspective, thus improving the detection performance.

Teufl *et al.* [130] extracted multiple metadata of the app as features, including last modification time, price, description, category, downloads, user ratings, package size, number of screenshots, package name information, version number, developer ID, and their contact details. Grampurohit *et al.* [71] and Wang *et al.* [136] combined the category information with existing permission- and API-based features to improve the detection performance. Unlike using the default category information, Gorla *et al.* [69] proposed CHABADA, which divides the collected apps into 29 categories by clustering their description information. They regarded that it is normal for a weather category app to use the API call *getLastKnownLocation()* to get location information, while other apps in this cluster calling this API may be treated as exceptions. This approach effectively combines descriptive information with software behavior to mine anomalous apps in various categories.

4) Graph Model

All of the above three types of features are represented in the form of strings that are easily changed by existing obfuscation techniques to bypass malware detection. Therefore, more and more research work attempts to improve the robustness of features while retaining the program semantics. Various graph models are proposed for malware analysis. According to the node properties, existing graph models can be categorized into function call graph (FCG), control flow graph (CFG), data flow graph (DFG), and user interface interaction graph (UIG). In addition, there are several customized graph models designed for specific requirements.

Table 2.1 lists different graph models and related information, the last column indicates the analysis grain, which is divided into three grades (i.e., coarse, medium, and fine) according to the analysis object.

Table 2.1: Descriptions of graph model based approaches

Graph Model	Typical Approaches	Node Property	Grain
FCG	Adagio [67], MaMaDroid [95], MIGDroid [77]	function name	medium
CFG	Centroid [46], GroupDroid [94], ADAM [128], SMART [96]	basic block	fine
DFG	DNADroid [51], PVCS [140]	statement	fine
UIG	ViewDroid [148], ResDroid [123], MassVet [47], SmartDroid [153]	view	coarse
PDG (Package Dependency Graph)	PiggyApp [154]	package	coarse
CDG (Class Dependency Graph)	DR-Droid [131], Droidlegacy [54]	class	coarse
ADG (API Dependency Graph)	DroidSIFT [149]	API	medium
HIG (Heterogeneous Information Network)	HinDroid [76]	API and app	medium

The fine grain model is the graph model in which the analysis object is the program statement. A typical work is the CFG constructed in Centroid [46], where each node denotes the basic block and each edge denotes the jump relationship between the blocks. However, DNADroid[51] focused on the analysis of DFG, where each node denotes the statement that contains data information, and each edge denotes the dependency relationship between statements.

The medium grain model is the graph model in which the analysis object is the function. A typical work is the FCG constructed in Adagio [67], where each node denotes a function and each edge denotes the call relationship between functions. Based on FCG, DroidSIFT [149] focused on the analysis of API calls, thus it constructed the API dependency graph, where each node denotes a API call and each edge denotes the dependency relationship between API calls. Both of the above two models belong to the isomorphic information network, meaning that the all the nodes have only one property. HinDroid [76] focused on the relationships between apps and proposed a heterogeneous information network , where each node denotes an API call or an app. The edge is constructed according to four conditions: 1) an app uses a API call; 2) two API calls belong to the same package; 3) two API calls belong to the same basic block; 4) two API calls use the same invocation methods.

The coarse grain model is the graph model in which the analysis object is the package, class or view. A typical work is the PDG constructed in PiggyApp [154], where each node denotes a package, and each edge denotes one of four different types of dependency relationships, including class inheritance, package homogeny, method calls, and member field references. Similarly, DroidLegacy [54] constructed the CDG, where each node denotes a class and each edge denotes the dependency relationships between classes. In addition, by analyzing the user interface, MassVet [47] constructed the UIG, where each node denotes the *activity* component and different types of dialogs, each edge denote different types of event messages, such as *onClick()*, *onFocusChange()*, and *onTouch()*.

2.3 Graph Embedding Techniques

After extracting the graphs, we must encode them in vector spaces before we can apply machine learning techniques. The embedding techniques based on representing graphs in vector spaces, while preserving their properties, have become widely popular. There are two types of representation learning.

The first is to encode nodes as low-dimensional vectors that summarize their structural roles in graphs. Perozzi *et al.* [105] proposed DeepWalk which first uses the random walks to generate node sequence as its context. Inspired by the skip-gram model [97], each node is regarded as a word and its representation is learned with a neural network. Then Grover and Leskovec [72] improved the DeepWalk model by proposing node2vec that uses second-order random walks to generate the node sequence. However, these approaches have a limitation, i.e., the structurally similar nodes will never share the same context if their distance is larger than the skip-gram window. Ribeiro *et al.* [113] proposed struc2vec, which uses a hierarchy to measure node similarity at different scales, and constructs a multilayer graph to encode structural similarities and generate the structural context for nodes. However, the above approaches cannot be directly applied to our work since their embedding results of the same graph are not in a consensus due to the using of random walks.

The second is to encode a graph as low-dimensional vectors instead of a node. Dai *et al.* [53] proposed structure2vec, which is based on the idea of embedding latent variable models into feature spaces and learning such feature spaces using discriminative information. Narayanan *et al.* [101] proposed graph2vec, which is also based on the skip-gram model for learning embedding similar to node2vec. The difference is that it views an entire graph as a document and the subgraphs around each node in the graph as words that compose the document. Even though such approaches can learn representations for graphs, they require a graph set as input and need model retraining to deal with the new coming samples.

2.4 Natural Language Processing for Android

With the development of NLP techniques, there are some approaches that analyze the Android-related contextual content to improve the analysis of relative tasks, such as permission analysis, privacy analysis, and malware analysis.

For the permission analysis, Rahul *et al.* [104] proposed WHYPER, which first extracts the requests for three permissions (i.e., READ_CONTACTS, READ_CALENDAR, and RECORD_AUDIO). Then, WHYPER leverages NLP and automates risk assessment of mobile apps by revealing discrepancies between app descriptions and their actual functionalities. The experiments on 581 samples demonstrate that WHYPER can achieve a precision of 82.8% and a recall of 81.5%. However, it can not deal with permissions that have no associated API calls, such as the permission RECEIVE_BOOT_COMPLETED. To handle the limitations of WHYPER, Qu *et al.* [108] proposed AutoCog, which can automatically assess description-to-permission fidelity of apps by extracting semantic information from the descriptions. The evaluation on 1,785 apps shows that the precision and recall of AutoCog are 92.6% and 92%, respectively.

As to the privacy policy analysis, Yu *et al.* [147] proposed PPChecker, which adopts NLP and program analysis techniques to automatically identify three privacy problems, the incomplete, incorrect, and inconsistent privacy policies. The experiment results on

1,179 apps demonstrate that 282 (23.6%) apps have at least one privacy problem. Slavin *et. al* [125] proposed a framework that detects the privacy violation based on a privacy-policy-phrase ontology and a set of mapping from API calls to policy phrases. The case studies on 501 top Android apps show 63 potential privacy policy violations.

For malware analysis, Gorla *et al.* [69] proposed CHABADA, which first groups the Android apps into clusters according to their description topics and then identify outliers in each cluster with respect to the API call usage. Zhu and Dumitras [157] proposed FeatureSmith, which engineers features automatically by analyzing the content of papers published in security conferences. Its detection accuracy is comparable to the performance of malware detectors that relies on manually engineered features.

Chapter 3

The Construction of Maximum Sensitive Subgraph Feature and Its Application in Piggybacked Apps Detection

3.1 Overview

Android smartphones have recently gained much popularity. Many Android app markets such as Google Play[20] and Anzhi Market[14] have been set up, where users can download various apps. The Android platform has become a major target of malware. According to Zhou and Jiang[156], piggybacking is one of the most common techniques utilized by malware authors to piggyback malicious payloads on popular apps to produce malware. About 86% of their collected 1,260 samples were piggybacked versions of legitimate apps with malicious payloads. The malware created through piggybacking is called a piggybacked app. A piggybacked app has two main parts, namely, the original benign code and the injected malicious payloads. Following the conventions described in [154], we use the term *carrier* to refer to the former and the term *rider* to refer to the latter.

Developing new malware from scratch is labor intensive, but malware authors can easily add a specific rider into various carriers through piggybacking techniques to quickly produce and distribute a large number of piggybacked apps. For example, members of the notorious malware family *geinimi* usually repackage themselves into various legitimate game apps, steal personal information and send it to a remote server. Typically, malware authors download paid apps from the official market, disassemble them, add malicious payloads, reassemble and submit the “new” apps to the official or alternative Android markets for free. The new piggybacked apps would entice smartphone users to download and install.

The rapid growth in the number of piggybacked apps poses great threats to smartphone users. To bypass existing malware detection approaches, malware authors create many different types of variants by constantly modifying the malicious source code. In addition, the increasingly sophisticated obfuscation technologies would change the structure of malicious code, making it more and more difficult to detect.

In this work, we detect Android piggybacked apps by utilizing the distinguishable invocation patterns of sensitive API calls between the rider and carrier. Sensitive API calls are the API calls that operate on sensitive data to perform malicious activities. It is worth noting that sensitive API calls constitute only a small portion of the whole Android API calls and they cannot be easily obfuscated by existing techniques whereas the names of user-defined functions are usually obfuscated as *a*, *b* or *c*.

To further understand the distinguishable invocation patterns, we establish two assumptions based on an empirical analysis of piggybacked apps.

Assumption 1: To perform its malicious task, the rider invokes more sensitive API calls than the carrier does.

Assumption 2: Generally, in the rider, the cohesion of sensitive API calls, which is measured by calling distances, is higher than that in the carrier.

By exploiting the two assumptions, we develop DAPASA, an approach to detect Android piggybacked apps through sensitive subgraph analysis. DAPASA consists of the following four steps.

1) DAPASA starts with the construction of a static function-call graph of a given app. It is a directed graph where nodes denote the functions invoked by the app and edges denote the actual calls among these functions.

2) To differentiate the maliciousness of different sensitive API calls, DAPASA calculates the sensitivity coefficients for each sensitive API through a term frequency-inverse document frequency (TF-IDF)-like measure.

3) DAPASA divides the static function-call graph into a set of subgraphs heuristically with sensitive API nodes and their nearby normal nodes. The subgraph with the highest sensitivity coefficient is selected as the maximum sensitive subgraph (SSG) to profile the most suspicious behavior of the given app.

4) Five features are constructed from the SSG. The feature sensitivity coefficient of the SSG (scg) and the feature total sensitive distance of the SSG (tsd) are used to measure the maliciousness and cohesion of sensitive API calls, respectively. In addition, three different types of sensitive motifs are exploited to further depict in a fine-grained manner the local invocation patterns of sensitive API calls. Finally, the five features are fed into machine learning algorithms to detect whether the app is piggybacked or benign.

DAPASA is evaluated on a large real-world dataset consisting of 2,551 piggybacked apps and 44,921 popular benign apps. The evaluation results show that DAPASA achieves good performance with a true positive rate of 95% and a false positive rate of 0.7%.

In summary, we make the following contributions in DAPASA.

(i) We propose two assumptions about the different invocation patterns of sensitive API calls between the rider and carrier in Android piggybacked apps. By exploiting these two assumptions, we construct a maximum sensitive subgraph to represent the entire call graph and profile the most suspicious behavior of the given app.

(ii) We propose five numeric features from the generated maximum sensitive subgraph. These features can not only be used for independent detection of piggybacked apps, but also have the ability to complement permission- and API-based approaches in the performance and explanation of the detection results.

(iii) We propose a TF-IDF-like measure to calculate the sensitivity coefficient of each sensitive API call based on the idea of TF-IDF. It can reduce the interference factors of the sensitive API calls that frequently occur in both benign and malicious apps.

The rest of this chapter is organized as follows. Section 3.2 introduces the construction of function call graph. Section 3.3 details the construction of maximum sensitive subgraph feature. Section 3.4 reports the evaluation results. Section 3.5 concludes this chapter.

3.2 Construction of Function Call Graph

Android apps are normally written in Java and compiled to dalvik code (DEX) stored in a *classes.dex* file. The compiled code and the required resources are packaged into an APK file. On the basis of existing disassemble tools (e.g., *apktool* [16]), we can obtain the dalvik code from the APK.

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

To retain the program semantics and be resilient to typical code obfuscation techniques, different kinds of effective graph models, including FCG [67], CFG [52, 51, 46], and user interface graph (UIG) [47, 123], are proposed. In our approach, we use FCG rather than CFG and UIG as our graph model to depict app behaviors because of two reasons. First, UIG is not suitable for similarity detection between malware samples since they usually have entirely different UIs. Second, although CFG is a fine-grained graph model that contains detail information of the basic blocks in methods, the extraction and analysis of CFGs is a time-consuming job that requires considerable computational resources. In addition, the results of related approaches [67] have proved that FCG contains enough semantic information to perform malware analysis.

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

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

Fig. 3.1 illustrates the FCG of an app called *corner23*. Thousands of nodes are usually found in a constructed FCG. The analysis of entire FCGs are neither effective (i.e., the malicious components constitute only a small portion) nor efficient (i.e., excessive number

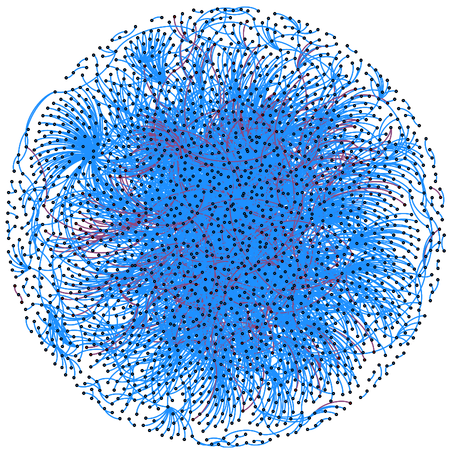


Figure 3.1: Function call graph of an app called *corner23*

of nodes and edges to analyze). Since Android malware usually invokes sensitive API calls that operate on sensitive data to perform malicious activities, we identify the sensitive API call nodes in FCGs. We employ the result reported by [111] to obtain a set of sensitive API calls. Specifically, [111] proposed *SuSi*, which is a novel machine-learning-guided approach, to identify *Sources* and *Sinks* directly from Android API calls. *Sources* are API calls that return sensitive data (e.g., *getDeviceId()* to obtain the IMEI of a phone), and *Sinks* are API calls that can use sensitive data as arguments (e.g., *sendTextMessage()* to send short messages). A total of 9,730 sensitive API calls are available and we use SS to denote the sensitive API call set.

It is worth noting that the widely-used third-party and advertisement libraries can introduce noises when analyzing malicious activities. To solve this problem, two filtering methods are applied. First, a list that contains widely-used library names provided by existing approaches [88, 87] is constructed. Second, a list of class names collected from 5,000 benign apps is also constructed. The class names on the two lists are regarded as noises and their corresponding subgraphs are removed from the subgraph set.

Although the two lists can work well for most apps, they are not sound for the class files whose names are obfuscated as a, b or c. To this end, we use the tool *Deguard* [41] to reverse the obfuscated names of given apps. Then we are able to remove the obfuscated class files if they are obtained in the above two lists.

3.3 Construction of Maximum Sensitive Subgraph Feature

After the construction of FCG, this section introduces the construction of maximum sensitive subgraph feature, which includes three steps: measuring the sensitivity coefficient of each sensitive API call, mining the maximum sensitive subgraph from the FCG, and constructing a feature vector from the maximum sensitive subgraph for each app.

3.3.1 Measurement of the Sensitivity Coefficient

The sensitivity coefficient is calculated to denote the maliciousness of a sensitive API call in performing malicious behavior. Given that several sensitive API calls are used in malware and benign apps, the measurement would be biased if only the coefficient of a sensitive API call is calculated with its frequency of occurrence in a malicious dataset, such as MIGDroid [77].

We propose a TF-IDF-like measure of the sensitivity coefficient of sensitive API calls that exploits the idea of TF-IDF [142]. To achieve this, 6,154 malicious apps are downloaded from VirusShare[28], and 44,921 benign apps in 26 categories, such as *Game*, *Personalization*, and *Weather*, are collected from Google Play and Anzhi Market. The benign apps have been uploaded to VirusTotal to ensure that they are benign according to the detection results provided by more than 50 anti-virus engines [29]. We use six terms of sensitive API call s to understand its distribution in our malicious and benign datasets.

- $mc(s)$: malicious count of s. It denotes the number of malware using s in the malicious dataset.
- $bc(s, c)$: benign count of s. It denotes the number of benign apps using s in category c.
- $mrt(s)$: ratio of $mc(s)$ to the total number of malware in the malicious dataset which is represented as p. $mrt(s)$ can be obtained with $mrt(s) = \frac{mc(s)}{p}$, where $p = 6,154$ in our work.
- $brt(s, c)$: ratio of $bc(s, c)$ to the total number of benign apps in category c which is represented as $q(c)$. $brt(s, c)$ can be obtained with $brt(s, c) = \frac{1+bc(s,c)}{q(c)}$.
- $mrk(s)$: rank number of $mrt(s)$ among all the sensitive API calls.
- $brk(s, c)$: rank number of $brt(s, c)$ among all the sensitive API calls in category c.

Table 3.1 shows several sensitive API calls with high mrts and their corresponding brts and brks in *Game*, *Personalization*, and *Weather* categories, respectively. We obtain three observations from Table 3.1.

1) Several sensitive API calls are used frequently in the malicious and benign datasets. For example, *openConnection()* and *connect()* are used to connect the Internet. Regardless of the category, their brks are very small.

2) Several sensitive API calls are used more frequently in the malicious dataset than in the benign dataset. An example is *sendTextMessage()*. Its mrk is 2, whereas its brks in all the three categories exceed 50.

Table 3.1: Several sensitive API calls' *mrt*s, *mrk*s, and their corresponding *brt*s, *brk*s in *Game*, *Personalization* and *Weather* categories.

Sensitive API Call	Malicious Dataset		Game		Personalization		Weather	
	mrt	mrk	brt	brk	brt	brk	brt	brk
notif()	0.746	1	0.201	23	0.212	4	0.465	14
sendTextMessage()	0.542	2	0.034	50	0.008	65	0.009	80
openConnection()	0.479	3	0.799	1	0.297	1	0.838	1
getDeviceId()	0.474	4	0.456	10	0.085	13	0.414	16
getLine1Number()	0.452	5	0.110	32	0.016	43	0.014	65
connect()	0.449	6	0.747	3	0.216	3	0.738	2
getInputStream()	0.344	9	0.263	18	0.051	23	0.516	11
getSubscriberId()	0.344	10	0.073	39	0.012	50	0.275	31
getConnectionInfo()	0.320	11	0.178	25	0.018	40	0.074	35
getSimSerialNumber()	0.285	14	0.050	44	0.010	58	0.014	64
getActiveNetworkInfo()	0.260	18	0.747	2	0.279	2	0.738	3
getLastKnownLocation()	0.217	21	0.418	13	0.079	14	0.595	8
requestLocationUpdates()	0.181	22	0.267	17	0.057	20	0.599	7
getCellLocation()	0.152	26	0.023	54	0.004	79	0.039	43

3) The brts and brks differ in the different categories. For example, in categories *Game* and *Weather*, nearly all sensitive API calls have higher brts than those in the category *Personalization*, which have lower than 0.3.

With these three observations, we consider the following questions to better understand our measurement of the sensitivity coefficient.

Q 1 ***If the mrt of a sensitive API call is high, will its sensitivity coefficient also be high?***

As illustrated in Table 3.1, the mrts of *openConnection()* and *getSimSerialNumber()* (used to obtain the user's SIM number) are 0.479 and 0.285, respectively. Does this mean *openConnection()* has a higher sensitivity coefficient than *getSimSerialNumber()*?

The answer is no. As noted in the first two observations, *openConnection()* is widely used in both malicious and benign apps because nowadays, most apps need to connect to the Internet. Meanwhile, *getSimSerialNumber()* occurs much more frequently in malicious apps than in benign apps because benign apps rarely need to have the SIM number. Intuitively, *getSimSerialNumber()* should have a higher sensitivity coefficient than *openConnection()*.

Q 2 ***Does an app that obtain location information by using getLastKnownLocation() appear suspicious?***

The answer is also no. As noted in the last observation, the brts in the three categories are different. For apps in the *Personalization* category, the API call's brt is only 0.079 and would reveal the location information of users. In the *Weather* category, the API call's brt is 0.595, and the API call is generally used to obtain weather information in the location of users. According to this discussion, the category information can be exploited in our measurement of sensitivity coefficients. For the same sensitive API call, its sensitivity coefficients in different categories would be different.

In text mining literature, TF-IDF is a numerical statistic intended to reflect how discriminating a term is to a document in a corpus. By utilizing the idea of TF-IDF for reference, we make the scs of a sensitive API call be in positive correlation with its mrt and in negative correlation with its brt. For sensitive API call s of an app that belongs to a specific category c, its sensitivity coefficient $scs(s)$ is calculated with Eq. (3.1).

$$scs(s) = mrt(s) \times \log \frac{1}{brt(s,c)}. \tag{3.1}$$

For example, the *Game* category has 3,505 apps, in which 2,801 apps use *openConnection()* and 174 apps use *getSimSerialNumber()*. Their scss are 0.047 and 0.371, respectively. Apparently, *getSimSerialNumber()* is more sensitive than *openConnection()*.

Table 3.2 shows the scss and $rank$s of the sensitive APIs. *sendTextMessage()* has

Table 3.2: Several sensitive API calls' *scs*s and their corresponding *rank*s in *Game*, *Personalization* and *Weather* categories.

Sensitive API Call	Game		Personalization		Weather	
	scs	rank	scs	rank	scs	rank
notif()	0.521	2	0.501	9	0.246	7
sendTextMessage()	0.792	1	1.130	1	1.102	1
openConnection()	0.047	46	0.252	21	0.037	43
getDeviceId()	0.161	20	0.508	8	0.181	12
getLine1Number()	0.432	3	0.807	2	0.839	2
connect()	0.057	43	0.299	16	0.059	35
getInputStream()	0.200	18	0.447	11	0.099	24
getSubscriberId()	0.391	4	0.657	3	0.192	11
getConnectionInfo()	0.240	9	0.561	5	0.362	4
getSimSerialNumber()	0.371	5	0.575	4	0.529	3
getActiveNetworkInfo()	0.033	51	0.144	32	0.034	44
getLastKnownLocation()	0.082	36	0.239	23	0.049	38
requestLocationUpdates()	0.203	17	0.232	24	0.063	32
getCellLocation()	0.249	8	0.364	12	0.214	8

the highest scs in all the three categories, given that it is frequently used by malicious apps and rarely used by benign apps. This condition reflects the common attack of stealthily sending SMS messages to premium numbers, thus allowing the owner of these numbers to earn money from the victims. Combined with sending SMS messages, the sensitive API calls utilized to obtain the user's privacy information, such as phone number (*getLine1Number()*) and SIM number (*getSimSerialNumber()*), would also have high coefficients. Unlike the previous ones, sensitive API calls used frequently both in malicious and benign apps, such as *openConnection()*, are assigned with low coefficients. The results show that the sensitivity coefficients calculated by the TF-IDF-like measure can reflect the maliciousness of sensitive APIs in different categories.

However, there are some apps that have no category information, especially the malware samples downloaded from VirusShare. We calculate the sensitivity coefficients of sensitive

API calls for such apps as:

$$scs(s) = mrt(s) \times \log \frac{1}{brt(s)}. \tag{3.2}$$

$brt(s)$ denotes the percent of apps in all benign apps using the sensitive API call s and it is obtained with Eq. (3.3), in which C denotes the set of all the benign categories.

$$brt(s) = \frac{1 + \sum_{c \in C} bc(s, c)}{\sum_{c \in C} q(c)} \tag{3.3}$$

3.3.2 Generation of SSG

Based on our proposed assumptions, FCG is divided into a set of subgraphs, and the subgraph that has the highest sensitivity coefficient is selected as the maximum sensitive subgraph (SSG), which can profile the suspicious behavior of the given app. SSG can be generated through the following steps.

Algorithm 1 highlights the step of generating the subgraph set with the input of the FCG of a given app and the sensitive API call set (SS). For each sensitive API node, a subgraph is constructed with its neighbor nodes in the FCG. The function $dis(v_k, v_i)$ returns the shortest path length from node v_k to node v_i. When calculating the distance between two nodes, the FCG is regarded as an undirected graph. The average shortest path length of the FCGs is generally from 3 to 5. When constructing subgraphs, the distances of normal nodes to the sensitive API node are less than or equal to 2.

Algorithm 2 highlights the step of selecting SSG from the SGS generated by algorithm 1. In SGS, two subgraphs that contain the same sensitive API nodes may exist. Algorithm 2 merges the subgraphs with the condition $V_s(sg_i) \cap V_s(sg_j) \neq \emptyset$ to ensure that one sensitive API node can only occur in one subgraph. $V_s(sg_i)$ denotes the set of sensitive

Algorithm 1 Generation of Subgraph Set (SGS)

```
Require: FCG $G = (V, E)$, $SS$
Ensure: $SGS$
 1: $SGS \leftarrow \emptyset$
 2: for each $v_i \in SS$ do
 3:     $V_i \leftarrow \emptyset$
 4:     for each $v_k \in V$ do
 5:         if $dis(v_k, v_i) <= 2$ then
 6:             $V_i = V_i \cup \{v_k\}$
 7:         end if
 8:     end for
 9:     $E_i = V_i \times V_i \cap E$
10:     $sg_i \leftarrow (V_i, E_i)$
11:     $SGS = SGS \cup \{sg_i\}$
12: end for
13: return $SGS$
```

Algorithm 2 Generation of Maximum Sensitive Subgraph (SSG)

```
Require: $SGS$
Ensure: $SSG$
 1: while $\exists sg_i, sg_j \in SGS, i \neq jandV_s(sg_i) \cap V_s(sg_j) \neq \emptyset$ do
 2:     $V_j = V_i \cup V_j, E_j = E_i \cup E_j$
 3:     $SGS = SGS \setminus \{sg_i\}$
 4: end while
 5: $scg(sg_j) = \sum_{s_i \in V_s(sg_j)} scs(s_i), 1 \leq j \leq m$
 6: $SSG = argmax_{sg_j \in SGS} (scg(sg_j))$
 7: return $SSG$
```

API calls in subgraph sg_i. Afterward, the sensitivity coefficient for each $sg_j \in SGS$ is calculated by adding all the sensitivity coefficient of the sensitive API nodes in the subgraph, and the subgraph with highest coefficient among all the subgraphs in SGS is selected as the SSG. If no sensitive API call exists in a given app, then it does not have an SSG.

Fig. 3.2 shows the extracted SSG of *corner23*. The SSG consists of 19 sensitive API calls and nearby normal nodes. By manually analyzing the code, we find that the SSG extracted from *corner23* is located in the most notorious module of the *geinimi* family. The module is used to collect users' sensitive information every five minutes, such as the device ID via *getDeviceId()* and the phone number via *getLine1Number()*.

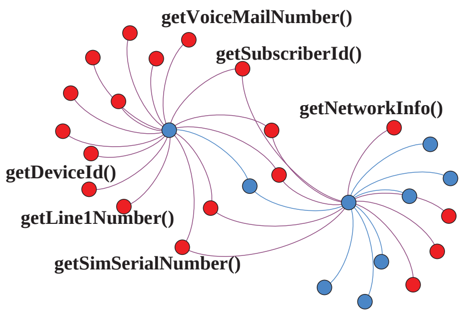


Figure 3.2: The generated SSG of app *corner23*

3.3.3 Construction of Features

By employing SSG, we construct a set of features from SSG based on our two proposed assumptions. The features fall into three fields to distinguish piggybacked apps from benign apps in different aspects. We randomly select 500 piggybacked apps and 500 benign apps, respectively, to determine if our features are able to distinguish them.

1) Sensitivity Coefficient of SSG ($scg(SSG)$**):** $scg(SSG)$ is defined to denote the maliciousness of SSG. As mentioned in assumption I, to perform its malicious task, the rider would make many sensitive API calls; thus, the maliciousness of SSG of a piggybacked app is higher than that of a benign app.

As illustrated in Fig. 3.3, the median of the coefficients of piggybacked apps is 1.341, which is higher than that of benign apps (0.444) because they have fewer invocations of sensitive API calls. This result proves that our assumption I is tenable. Obviously, $scg(SSG)$ can effectively distinguish piggybacked apps from benign ones.

2) Total Sensitive Distance of SSG ($tsd(SSG)$**):** As mentioned in assumption II, the cohesion of sensitive API calls in the rider is generally higher than that in the carrier. We use $tsd(SSG)$ to denote the cohesion of sensitive API nodes in SSG, which is measured by the calling distances between sensitive API nodes.

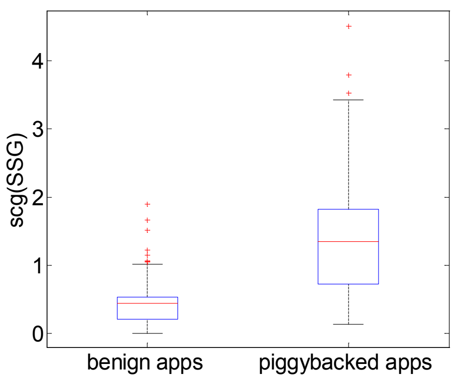


Figure 3.3: $scg(SSG)$ for benign and piggybacked apps

$tsd(SSG)$ can be obtained with Eqs. (3.4) and (3.5), in which $sd(s_i)$ denotes the average distance of sensitive API node s_i to the other sensitive API nodes in SSG.

$$tsd(SSG) = \sum_{s_i \in V_s(SSG)} sd(s_i) \tag{3.4}$$

$$sd(s_i) = \frac{1}{|V_s(SSG)| - 1} * \sum_{s_j \in V_s(SSG), j \neq i} \frac{1}{dis(s_i, s_j)} \tag{3.5}$$

As illustrated in Fig. 3.4, the median of $tsd(SSG)$ of piggybacked apps is 1.875, which is even higher than the upper quartile of benign apps. This result indicates that assumption II is tenable. Thus, the feature $tsd(SSG)$ is effective to distinguish piggybacked apps from benign ones.

3) Total Number of Sensitive Motif Instances in SSG ($tnsm(SSG)$): We have attempted to obtain a more detailed view of the invocation patterns between sensitive API nodes and normal nodes. An invocation pattern reflects one malicious behavior of an app, which can be depicted by a motif. Network motifs are defined in terms of connectivity-patterns that appear much more often than expected from pure chance [98, 114, 139]. Specifically, they occur at a higher frequency than what is expected from an ensemble of randomized graphs with an identical degree structure. Given that no mutual edges exist

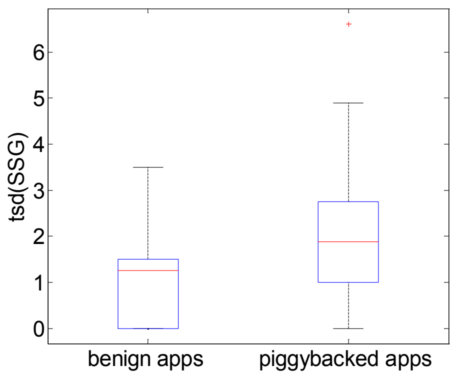
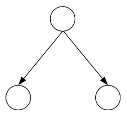
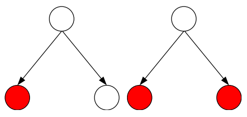
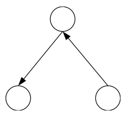
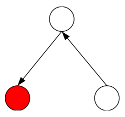
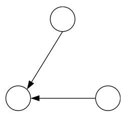
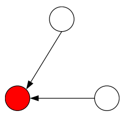
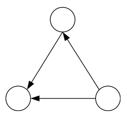


Figure 3.4: $tsd(SSG)$ for benign and piggybacked apps

Table 3.3: Three node motifs and their corresponding sensitive motifs

Three-node Motifs			Sensitive Motif	
Index	Pattern	Z-score	Index	Pattern
motif-1		1.349	sensitive-motif-1	
motif-2		1.356	sensitive-motif-2	
motif-3		1.308	sensitive-motif-3	
motif-4		-0.499		

in SSG, four three-node motifs are present. The four three-node motifs and their average Z-score values in our samples are shown in Table 3.3 with the help of gtrieScanner [21]. The higher the Z-score is, the more significant the three-node pattern is as a motif. The Z-score of motif-4 is less than 0, which means that it rarely occurs in SSG. Thus, it is ignored in our computation.

Sensitive motifs are defined in this work as significant motifs that contain at least one

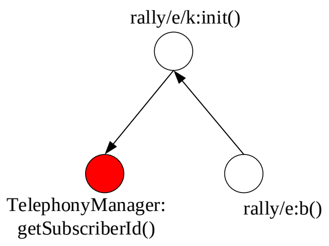


Figure 3.5: An instance of sensitive motif-2 in SSG

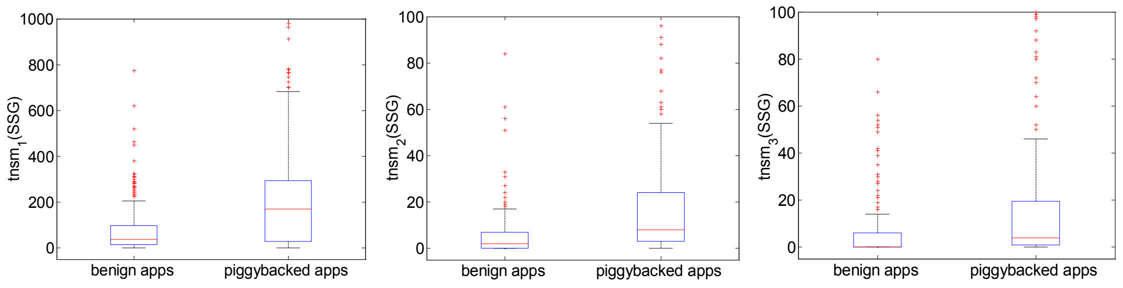


Figure 3.6: $tnsm(SSG)$ for benign and piggybacked apps

sensitive API node. They are shown in Table 3.3. For example, the instance of sensitive motif-2 in Fig. 3.5 denotes the malicious behavior of obtaining the unique subscriber ID number by using an object *rally/e* and invoking the *getSubscriberId()* API.

Under assumptions I and II, because of the larger number and higher cohesion of sensitive API calls in the rider than in the carrier, more instances of sensitive motif-1 occur in SSG. In addition, in the rider, the sensitive API calls are invoked by many user-defined threatening functions, which cause many instances of sensitive motif-2 and sensitive motif-3. We use $tnsm_k(SSG), k = 1, 2, 3$, to denote the total number of sensitive motif-k instances in SSG. Fig. 3.6 illustrates $tnsm_k(SSG)$ for our benign and piggybacked apps, which demonstrates that for all the three types of sensitive motifs, the corresponding $tnsm_k(SSG)$ for piggybacked apps are higher than those for benign apps.

The features constructed from the SSGs of piggybacked apps differ significantly from those of benign apps. DAPASA embeds the above five features into a feature space to automatically classify novel apps as piggybacked apps or not. The feature space is represented as follows:

$$\begin{pmatrix} scg(SSG) \\ tsd(SSG) \\ tnsm_1(SSG) \\ tnsm_2(SSG) \\ tnsm_3(SSG) \end{pmatrix}. \tag{3.6}$$

3.4 Evaluation of DAPASA

To evaluate the effectiveness of our approach, we first introduce the dataset and the metrics (see Section 3.4.1 for details). We then evaluate the detection performance and the runtime overhead of our approach based on the dataset and compare the result with that of three baseline approaches (see Section 3.4.2 and Section 3.4.3 for details). Afterward, we analyze the effectiveness of our features and how they complement the permission- and API-based approaches (see Section 3.4.4 for details). Finally, we discuss the false positives and the resilient of our approach to obfuscation techniques (see Section 3.4.5 for details).

3.4.1 Study Setup

Our approach is evaluated on a large real-world dataset that consists of Android benign apps and piggybacked apps. The set of piggybacked apps contains 2,551 apps in 15 families. All the apps are piggybacked apps according to [156]. A total of 1,062 of the apps are downloaded from the Android Malware Genome Project, which is widely used as a benchmark dataset for malware detection. We collect 1,489 more piggybacked apps that belong to the piggybacked families [156] from VirusShare. An overview of the piggybacked apps in our dataset is given in Table 3.4.

The set of benign apps consists of two parts; one is collected from Google Play and

Table 3.4: Descriptions of the piggybacked apps

Family	#apps	Family	#apps
adrd	55	fakeinstaller	769
anserverBot	187	geinimi	94
basebridge	122	gingermarster	350
beanbot	8	golddream	46
bgserv	9	hippoSMS	13
droiddream	49	jifake	41
droiddreamlight	46	pjapps	66
droidkungfu	696	**Total**	**2,551**

Table 3.5: Descriptions of the benign apps

Google Play Apps			Anzhi Market Apps		
ID	Category	#apps	ID	Category	#apps
GA	Business	491	AA	Communication	1,122
GB	Comics	497	AB	Finance	2,177
GC	Communication	622	AC	Music& Audio	1,307
GD	Education	577	AD	News Reading	2,400
GE	Entertainment	798	AE	Office Work	4,970
GF	Finance	492	AF	Shopping	4,837
GG	Game	3,505	AG	Social	3,265
GH	Lifestyle	789	AH	System Tools	3,150
GI	Medical	374	AI	Themes Desktop	7,962
GJ	Personalization	732	AJ	Weather& Travel	1,730
GK	Photography	491		**Total**	**32,920**
GL	Productivity	569			
GM	Shopping	377			
GN	Social	630			
GO	Tools	625			
GP	Weather	432			
	Total	**12,001**			

Table 3.6: Descriptions of the used metrics

Term	Abbr.	Definition
True Positive	TP	#malicious apps classified as malicious apps
True Negative	TN	#benign apps classified as benign apps
False Negative	FN	#malicious apps classified as benign apps
False Positive	FP	#benign apps classified as malicious apps
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

contains 12,001 apps in 16 categories, and the other one is collected from Anzhi Market and contains 32,920 apps in 10 categories. Table 3.5 shows the descriptions of apps from Google Play and Anzhi Market. All the apps have been checked by VirusTotal [29] to ensure that each of them is benign. Over 50 anti-virus softwares programs, such as ESET-NOD32 [19] and McAfee [24], are available in VirusTotal; these software programs are based on a signature database. They are useful for known malware but less effective for unknown ones.

The metrics used to measure our detection results are shown in Table 3.6. The goal of any malware detection research is to achieve a high value for TPR and a low value for FPR.

3.4.2 Piggybacked App Detection

1) Detection Performances with Four Classifiers: Four different classifiers are employed to evaluate our approach. These classifiers are Random Forest [43], Decision Tree (C4.5) [120], k-NN(k=1) [32] and PART [64]. All the 49,921 benign apps and 2,551 piggybacked apps are mixed together. After the extraction and analysis of the SSGs with our approach,

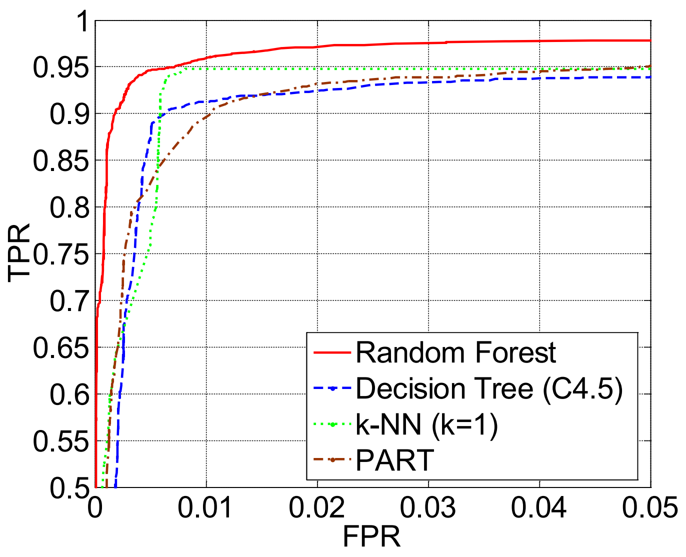


Figure 3.7: Detection performance with four different classifiers

each app is first represented as a feature vector. Then the classification labels of the known piggybacked apps in training dataset are attached with 1 while the labels of the known benign apps are attached with -1 so that the classifiers can understand the discrepancy between piggybacked apps and benign apps. Once the feature vectors with classification labels for the training samples are generated, four classifiers can be trained with the four machine learning algorithms. After that, the feature vector of a new sample without classification label is fed into the classifiers to detect whether it is piggybacked or benign. Our dataset is evaluated via tenfold cross validation.

The detection performance is shown in Fig. 3.7. The Receiver Operating Characteristic (ROC) curves indicate that all four classifiers can achieve a high value for TPR and a low value for FPR. In particular, Random Forest performs best among four classifiers. With Random Forest, the detection performance yields a TPR of 0.950 at an FPR of 0.007, and the AUC is 0.99.

Two main reasons explain the best performance of Random Forest in the current study's dataset. First, Random Forest is an ensemble classifier that uses out-of-bag errors as an estimate of the generalization error to improve its performance, whereas the other three classifiers are base classifiers. Second, as introduced in the work of Breiman[43], Random Forest does not result in overfitting as more trees are added but produces a limited value of the generalization error. Therefore, in this work, Random Forest is selected as the classifier in subsequent experiments.

2) Comparison with Three Baseline Approaches: In this section, DAPASA is compared with three baseline approaches proposed by Wang *et al.* [137], Aafer *et al.* [31], and Gascon *et al.* [67]. The descriptions of the three baseline approaches are shown below.

- Wang *et al.*[137] proposed an approach for malware detection based on *requested permissions*, which are security-aware features that restrict the access of apps to the core facilities of devices.
- Aafer *et al.*[31] proposed an approach for malware detection based on *APIs* that have more fine-grained features than permissions because each permission governs several API calls. Furthermore, API level information conveys more substantial semantics about the app than permissions [31].
- Gascon *et al.*[67] proposed an approach for malware detection based on *embedded call graphs*, which model the structural composition of a code and reflect the logic semantics of the app. The call graph is more robust against certain obfuscation strategies than the requested permissions and API calls.

The detection performance of our approach and the three baseline approaches in our dataset is illustrated in Fig. 3.8. The AUC values of our approach and API-based approach [31] are both 0.99, which indicates that our approach has a similar detection performance with API-based approach. Moreover, our approach outperforms the other two baseline

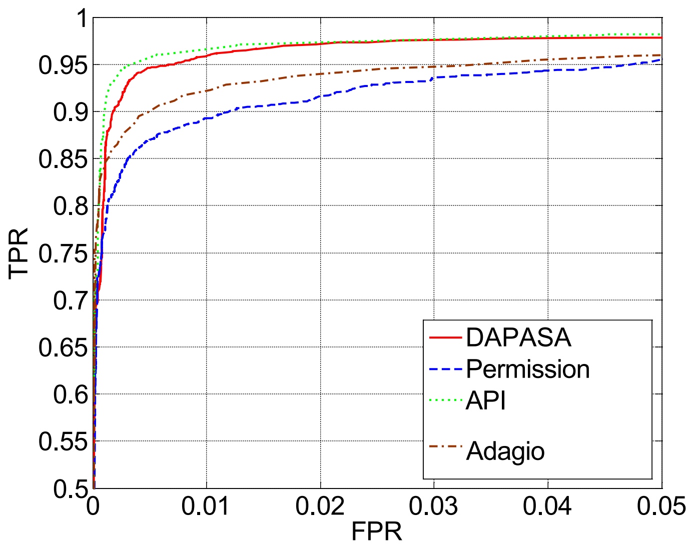


Figure 3.8: Detection performance for DAPASA and three baseline approaches

approaches [137, 67], of which the AUC values are 0.983 and 0.986, respectively. In particular, our approach contains only five numeric features; the three approaches use 88 permission-based features, 680 API-based features, and 32,768 graph-based features, respectively.

3.4.3 Analysis of Run-time Overhead

Our approach consists of three main procedures when analyzing a new app.

(i) **De-compilation.** The app file is disassembled to generate the dalvik code, and FCG is constructed.

(ii) **Graph analysis.** The FCG is divided into a set of subgraphs, and the SSG with the highest sensitivity coefficient is selected.

(iii) **Feature construction.** Five numeric features are constructed from the generated SSG.

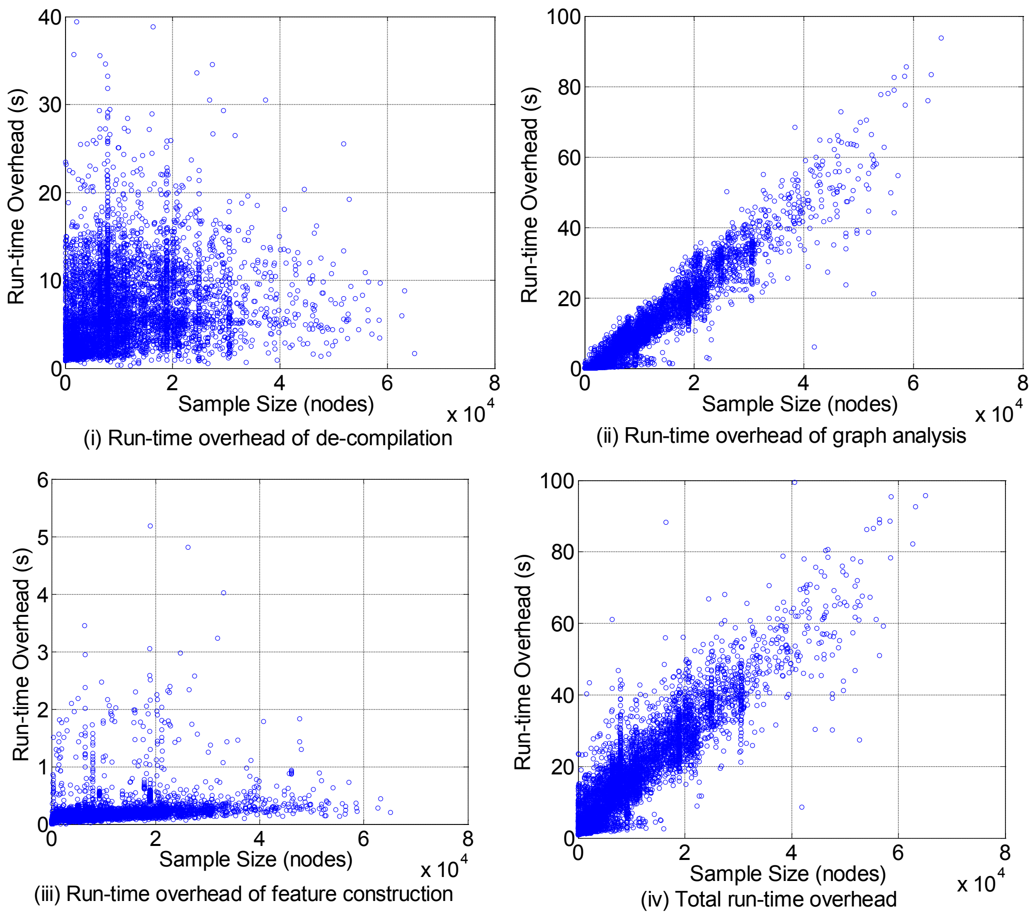


Figure 3.9: Run-time overhead of DAPASA

The run-time overheads of the three main procedures and their total run-time overhead are illustrated in Fig. 3.9, in which the *x*-axis shows the sample size (number of nodes) per app in our dataset and the *y*-axis shows the run-time overhead of the corresponding procedure.

Four observations are obtained from Fig. 3.9.

(1) The run-time overhead of de-compilation is not related to the sample size. This result is consistent with the truth that the complexity of de-compilation has a positive correlation with the logic of the source code for a given app rather than the sample size [80].

(2) The run-time overhead of graph analysis roughly scales linearly with the sample size. As introduced in algorithm 1, the time complexity is $O(m \times n)$, where m denotes the number of invoked sensitive API nodes and n denotes the size of the call graph.

(3) The run-time overhead of feature construction is not related to the sample size. In our approach, SSG is generated to represent the entire call graph. Therefore, the run-time overhead of feature construction scales with the size of SSG rather than the size of the call graph.

(4) The total run-time overhead of analyzing a given app has positive relation with sample size. It is mainly affected by the procedure of de-compilation with a relatively small sample size. With the increase in sample size, the total run-time overhead is mainly affected by the graph analysis procedure. On the average, less than 16s is consumed to complete the analysis for most apps in our dataset.

The comparison of the run-time overheads of our approach and the three baseline approaches is illustrated in Fig. 3.10. DAPASA consumes 1.8s and 4.6s less time than the approach of Gascon *et al.* [67] in graph analysis and feature construction, respectively. The smaller run-time overhead is due to the following reasons.

First, for the graph analysis procedure, in the approach of Gascon *et al.* [67], a hash-value is calculated for each node in the graph. Analyzing all the nodes consumes more time than our approach does because our approach only focuses on the analysis of sensitive API nodes.

Second, for the feature construction procedure, in the approach of Gascon *et al.* [67], a feature map is inspired by graph kernels, which allows for embedding call graphs in a vector space. However, our approach generates SSG to represent the entire call graph. Hence, computational complexity is reduced effectively.

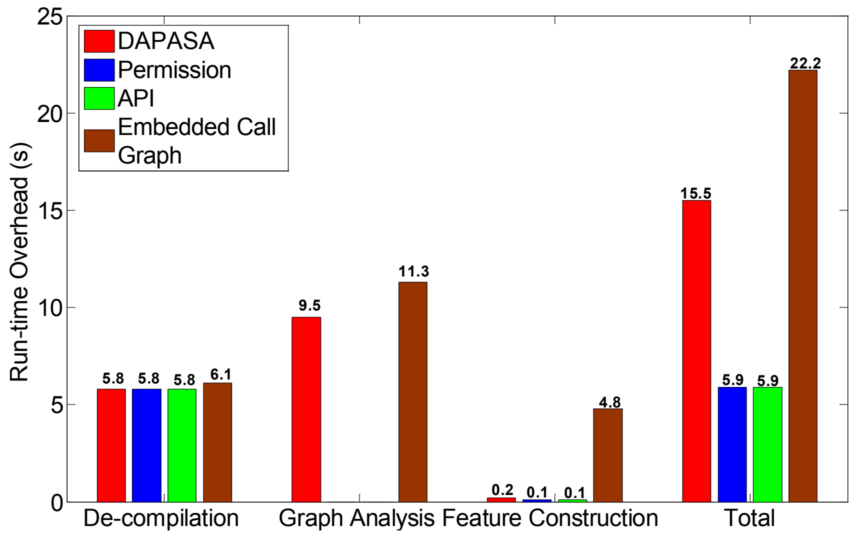


Figure 3.10: Comparison results of run-time overhead

The approaches of Wang *et al.*[137] and Aafer *et al.*[31] do not have the graph analysis procedure. Therefore, they are faster than DAPASA and the approach of Gascon *et al.* [67], which are based on the analysis of the call graph. Permission- and API-based approaches usually produce only a small run-time overhead, and they are efficient and scalable. However, the features of permissions and API calls are coarse-grained. For example, malicious apps may request the exact same permissions that are requested by benign apps.

3.4.4 Analysis of Features

In this work, we propose three different types of features, namely, scg, tsd, and $tnsm$ (consisting of $tnsm_1$, $tnsm_2$ and $tnsm_3$) to distinguish the SSGs existed in piggybacked apps from those existed in benign apps. As mentioned before, each of them has a fairly good ability to detect piggybacked apps in different aspects, such as maliciousness and cohesion of sensitive APIs. In this section, different combinations of features are evaluated in the same dataset to determine whether each feature is significant for the detection

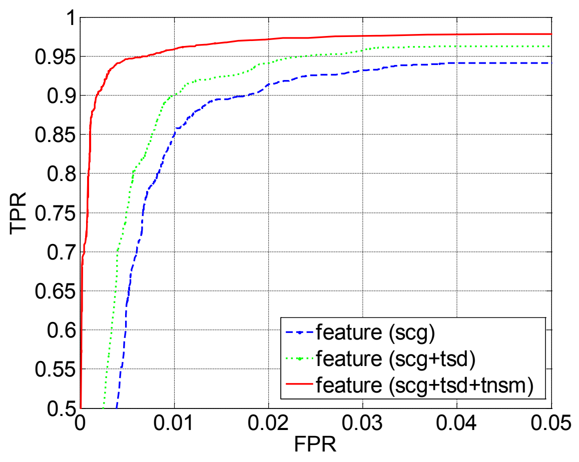


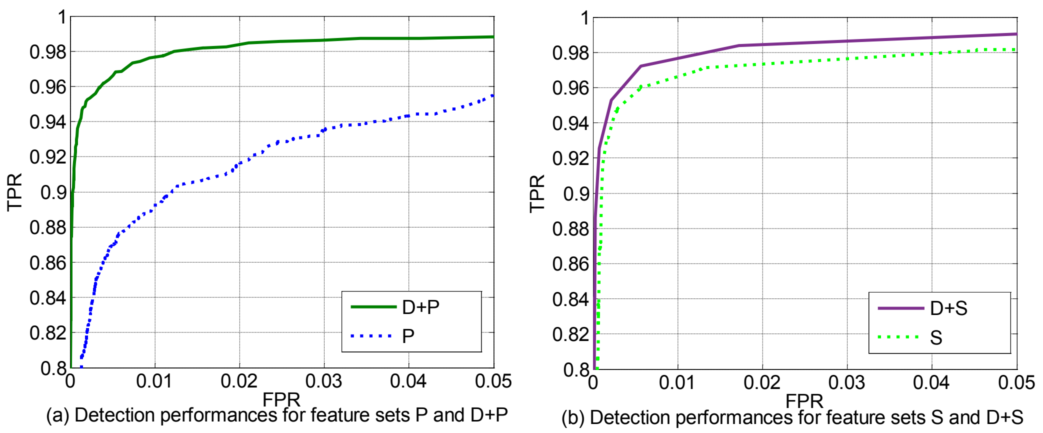
Figure 3.11: Detection performances with different feature combination

performance. Only scg is initially used as our feature. Afterward, the other two types of features are added to our feature space successively.

As illustrated in Fig. 3.11, the ROC curves with different feature combinations show that every additional feature effectively improves the detection performance. The TPR reaches nearly 0.85 with an 0.01 FPR using only scg, and it is improved by 0.05 and 0.061 by adding tsd and $tnsm$. The improvements of TPRs demonstrate that each proposed feature has significant contributions for piggybacked app detection.

Five features are constructed from a new perspective of the invocation structure. We combine five features with the permission-based features proposed by Wang *et al.* [137] and API-based features proposed by Asfer *et al.* [31], respectively. The detection performances of the four different feature sets are illustrated in Fig. 3.12, in which *P* denotes the 88 permissions, *S* denotes the 680 API calls, *D*+*P* denotes the combination of our five features with permissions, and *D*+*S* denotes the combination of our five features with API calls.

Moreover, the contribution degrees of our five features are evaluated with three different types of metrics, namely, chi-square statistic [106], OneR classifier [75], and information



(a) Detection performances for feature sets P and D+P (b) Detection performances for feature sets S and D+S

Figure 3.12: Detection performances for different feature sets

Table 3.7: Feature ranking of our features in the feature *D+P* and feature *D+S*

Feature	Chi-squared Statistic		OneR Classifier		Information Gain	
	D+P	D+S	D+P	D+S	D+P	D+S
scg	1	1	1	1	1	1
tsd	3	3	3	3	4	3
$tnsm_1$	2	2	2	2	2	2
$tnsm_2$	9	8	13	15	8	9
$tnsm_3$	8	6	4	5	7	7

gain [110], for the two combined feature sets (*D+P* and *D+S*) containing 93 and 685 features, respectively. The result in Table 3.7 shows that the five features have better contributions to classify piggybacked apps than most permission-based and API-based features especially scg and $tnsm_1$.

3.4.5 Discussions

In this section, we first inspect the reasons of the generation of false positive instances. Then the ability of DAPASA to fight against obfuscation attacks is discussed.

1) Discussions on TPR and FPR: The experiments show that our approach achieves good performance with a TPR of 95% and an FPR of 0.7%. Manual analysis of the SSGs of our piggybacked apps shows that DAPASA achieves a 100% detection rate in several families, such as *geinimi*. The invocation patterns of sensitive API calls in the generated SSGs for all the *geinimi* samples are exactly the same as that of the example introduced in Section 3.3.2. However, the TPRs are lower than 92% in several families, such as *droidkungfu*, which is considered one of the most sophisticated Android malware. *droidkungfu* is piggybacked and distributed in the forms of legitimate apps. Several samples implement their malicious functionalities in native code (instead of the previously davilk code based on Java). In this work, the native code is ignored, thus resulting in the lower accuracy for such families.

Although the TPR is impressive, the FPR is 0.7% which means that more than 300 benign apps are incorrectly classified as piggybacked apps. Two main reasons explain the incorrectly classified samples. First, with the help of LibD [88] we are able to remove most nodes invoked by third-party libraries with a string matching algorithm. However, covering all third-party libraries is still a challenge. Second, there are several extreme cases which are repackaged with only one sensitive API (*sendTextMessage*). When these extreme cases are placed in the training dataset, the benign apps using the same sensitive API as the extreme cases do might be classified as piggybacked. For example, in the *Game* category, sending a message to a premium number to raise money is a legitimate payment method for unlocking game features, and the apps that use this method would be incorrectly classified.

2) Resilience of Sensitive Subgraph: We evaluate the resilience of our constructed sensitive subgraph to two main types of obfuscation techniques: a) typical obfuscation techniques such as class renaming, inserting of useless instructions; b) advanced obfuscation techniques such as reflection techniques, encryption packer and native code.

- Renaming: We initially evaluate the resilience of sensitive subgraph to renaming user-defined functions [132] by using *Proguard* [26] 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 DAPASA because they do not change the FCG structure.

- Control flow obfuscation: We evaluate the resilience of sensitive subgraph to control flow obfuscation techniques, which will change the FCG structure by inserting or deleting some useless methods of a given app. For this purpose, we apply DAPASA to ten apps obfuscated by the popular Android obfuscator, *DashO* [17], which can adopt control flow obfuscation techniques. Results show that the SSGs 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 constructed feature vectors will be slightly affected when the inserted or deleted method nodes have invocation relations with the sensitive API call nodes.

- Reflection: Reflection techniques [112] 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* [86], 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*.

- Packer: Packers, such as *APKProtect* [15] and *Bangcle* [39], can protect apps by using encryption techniques to hide the actual Dex code. To address the limitations of packer usage, we use *PackerGrind* [144, 151], which is a novel adaptive unpacker system, to recover the actual Dex files. Then, our approach can be applied to the extracted Dex files.

- Native code: Malware can use native code 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* [124], which is an open-source binary analysis framework, to construct the FCG of the native code. Then, we could apply our feature extraction approach on the constructed FCG.

In summary, the proposed sensitive subgraph can be resilient to typical obfuscation techniques, and can handle advanced obfuscation techniques by leveraging existing tools.

3.5 Brief Summary

In this chapter, we propose DAPASA that focuses on piggybacked app detection through maximum sensitive subgraph analysis. First, two assumptions are proposed to better profile the differences between the rider and carrier in piggybacked apps with respect to the invocation patterns of sensitive API calls. Second, an SSG is generated for each app to profile its most suspicious behavior. Third, five features are constructed from the SSG and fed into machine learning approaches to detect piggybacked apps. Extensive evaluation results demonstrate that our approach achieves an impressive detection performance with only five numeric features which bring three advantages. First, our approach outperforms the state-of-the-art approaches with less features. Second, our approach provides better explanations of detection results than permission- and API-based approaches. Third, our approach even complements permission- and API-based approaches with the combination of our features from a new perspective of the invocation structure.

Chapter 4

The Construction of Frequent Sensitive Subgraph Feature and Its Application in Familial Identification

4.1 Overview

Chapter 3 proposes the maximum sensitive subgraph feature and applies it in piggybacked apps detection. However, the analysis of each malware sample requires ample time [150]. Hence, the sheer number of malware samples overwhelms malware analysis systems. The majority of new malware samples are polymorphic variants of known malware [156, 61]. 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, as introduced in Chapter 3, the accurate separation of malicious components and the legitimate part from the majority of Android malware, which are repackaged popular apps, is nontrivial [154, 54, 135]. 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 [90] and sensitive path [146].

```
.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 4.1: Different implementations of the same functionality in two malware samples within *geinimi* family

Second, polymorphic variants of Android malware that belong to the same family perform the same malicious activities with different implementations. Therefore, such malware can easily evade existing classification solutions [48, 65] that seek an exact match of a given specification. For example, Listing 4.1 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 first distill program semantics into FCG representation. Then, we propose two key techniques to solve the challenges 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.

To represent common malicious behaviors shared by malware samples within the same family, we construct frequent sensitive 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*. FalDroid consists of the following three steps.

1) Preprocessing: Based on the FCG model, FalDroid constructs a sensitive API related graph (SARG) for each app, and assigns different weights to each sensitive API call to denote its corresponding importance to each malware family.

2) Fregraph Generation: To easily locate the common functionalities of different malware samples and reduce the complexity of graph similarity calculation, the SARG is initially divided into a set of subgraphs using community detection algorithms. Using subgraph matching and clustering techniques, the sensitive subgraphs used by most samples in one family are defined as the **fregraphs** of a specific family.

3) Feature Construction: 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.

In summary, our major contributions include:

(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 malware 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 rest of this chapter is organized as follows. Section 4.2 introduces the construction of frequent sensitive subgraph feature. Section 4.3 describes the usages of fregraphs. Section 4.4 details the experimental results of FalDroid and Section 4.5 summaries this chapter.

4.2 Construction of Frequent Sensitive Subgraph Feature

4.2.1 Preprocessing

The *Preprocessing* stage constructs the basic behavior model for each app, and it contains two 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, 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)** by retaining only sensitive API call nodes and their parent nodes. Therefore, the malicious behaviors of the apps are maintained, whereas the complexity of the graph models are reduced.

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 $F = \{f_j | 1 \leq j \leq m\}$ denotes the set of all families, and m denotes the number of families.

Table 4.1: Six sensitive API calls' $totalNum$ and their corresponding num, per and w in three families ($allNum = 8,407$)

Sensitive API	*totalNum*	*geinimi* (*falNum=105*)			*plankton* (*falNum=896*)			*droidkungfu* (*falNum=736*)		
		num	*per*	*w*	*num*	*per*	*w*	*num*	*per*	*w*
getDevice SoftwareVersion()	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

We collect 8,407 malware samples in 36 families from Virusshare[28] for evaluation. Table 4.1 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 are used by most malware samples. For example, *getDeviceId()* is used by all samples in the three families.

The two observations indicate that the weight of a sensitive API call in one family should be positively related with its per and should be negatively related with its $totalNum$. By borrowing the idea of TF-IDF [142], 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:

$$w(s, f) = per(s, f) * \log \frac{allNum}{totalNum(s)}. \tag{4.1}$$

Table 4.1 shows that the weight of *sendTextMessage()* is 0.567 in the *geinimi* family, whereas that of *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 SARG: Thousands of nodes are usually found in the graph of an app. Analyzing the entire graph 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. (4.2) and (4.3), 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.

$$V_g = \{v_j | \exists v_i \in V_s, 0 < dis(v_j, v_i) < n, v_j \in V\} \tag{4.2}$$

$$V' = V_s \cup V_g, E' = (V' \times V') \cap E \tag{4.3}$$

In general, the size of SARG is reduced by approximately 72% compared with that of the original FCG. Fig. 4.1 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

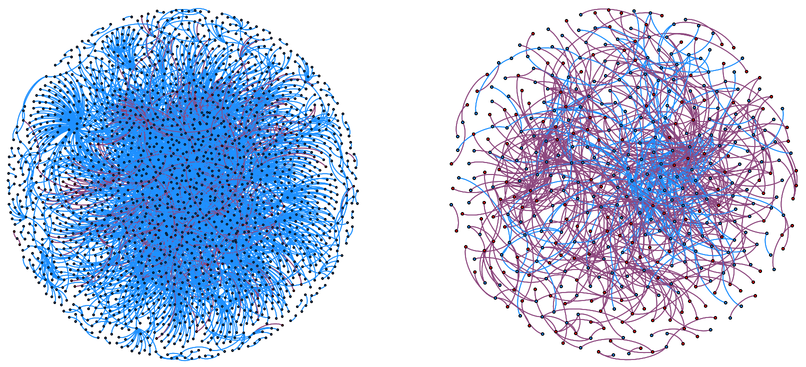


Figure 4.1: The original FCG (left) of a *geinimi* sample and its generated SARG (right)

and blue nodes denote general nodes. The red edges indicate that their callee functions are sensitive API call nodes.

4.2.2 Generation of Fregraph

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 and a weighted-sensitive-API-call-based graph matching approach to calculate the similarity between subgraphs generated with community detection algorithms.

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. 4.2 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

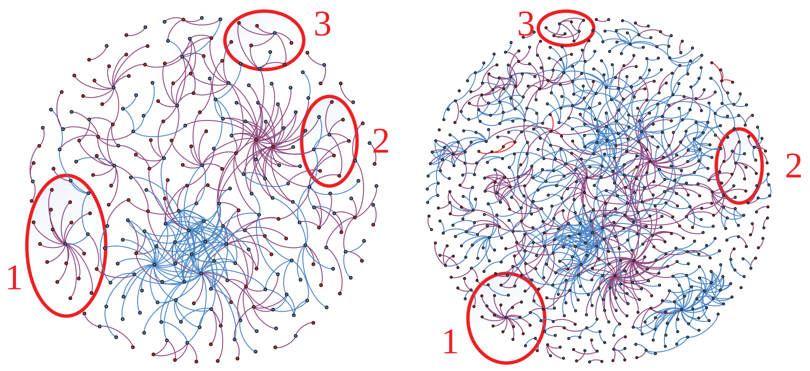


Figure 4.2: SARGs of two malware samples in the *geinimi* family. Three similar subgraphs marked with red circles indicate the similar behaviors.

direct identification of similar subgraphs from SARGs is inefficient because the graph isomorphism problem is NP complete [134]. 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 [102, 49], 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 [50, 107] have demonstrated that FCG is a typical network with community structures. Software functions in one community structure 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* [118], *fast greedy* [49], *fast partitioning* [109], and *multilevel* [42], to divide SARGs into a set of subgraphs. We implement the algorithms using *Networkx* [22], which is a package for computation of complex networks. We select *infomap* [118] 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

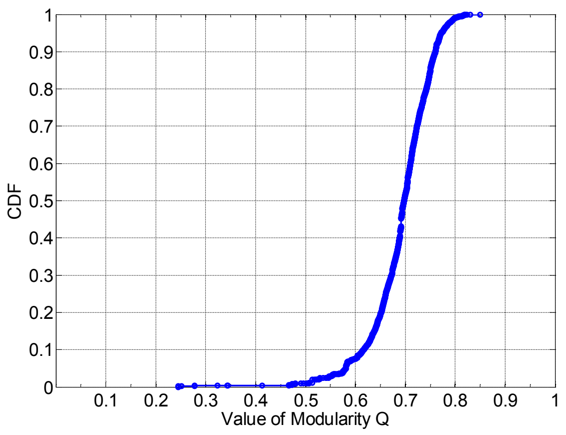


Figure 4.3: CDF of modularity Q with *infomap* algorithm

matching.

Newman and Girvan [102] 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. 4.3 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. It is a subgraph obtained from SARG using the community detection algorithm which 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. (4.4), to denote its importance to f. $V_s(sg)$ is the set of sensitive API call nodes in sg.

$$w(sg, f) = \sum_{v_i \in V_s(sg)} w(v_i, f) \tag{4.4}$$

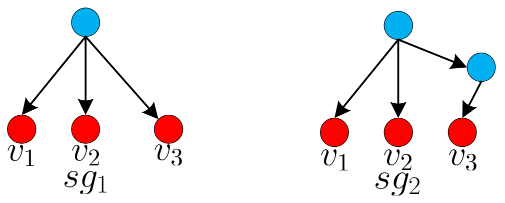


Figure 4.4: Two subgraph examples sg_1 and sg_2 in family f

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. 4.4 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 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. (4.5), and its size is $t \times t, t = |V_s(sg_1) \cup V_s(sg_2)|$. In Eq. (4.5), 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.

$$Matrix_k[i,j] = \begin{cases} dis'(v_i, v_j) & v_i, v_j \in V_s(sg_k) \\ \infty & otherwise \end{cases} \tag{4.5}$$

For the two subgraphs presented in Fig. 4.4, 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. (4.6). 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. (4.7).

$$\overrightarrow{vec(v_i, sg_k)} = \langle \frac{1}{Matrix_k(i,1)}, \ldots, \frac{1}{Matrix_k(i,t)} \rangle \tag{4.6}$$

$$ns(v_i) = cos(\overrightarrow{vec(v_i, sg_1)}, \overrightarrow{vec(v_i, sg_2)}) \tag{4.7}$$

The vectors of v_1 in the two subgraphs presented in Fig. 4.4 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

Algorithm 3 Clustering of Sensitive Subgraphs

```
Require: $SG_f$   // $SG_f$ denotes the set of sensitive subgraphs in family $f$.
    $\epsilon = 0.8$   // $\epsilon$ denotes the similarity threshold value.
Ensure: $C$   // $C$ denotes the set of output clusters and each cluster contains a set of
    similar sensitive subgraphs.
 1: $p = 1, c_1 = \{sg_1\}, C = \{c_1\}$
 2: for each $sg_{i,i \neq 1}$ in $SG_f$ do
 3:     $c' = argmax_{c_j \in C}\, \overline{sim_f}(sg_i, c_j)$
 4:     if $\overline{sim_f}(sg_i, c') \geq \epsilon$ then
 5:         $c' = c' \cup \{sg_i\}$
 6:     else
 7:         $p = p + 1, c_p = \{sg_i\}, C = C \cup \{c_p\}$
 8:     end if
 9: end for
10: return $C$
```

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:

$$sim_f(sg_1, sg_2) = \frac{\sum_{v_i \in V_s(sg_1) \cap V_s(sg_2)} (w(v_i, f) * ns(v_i))}{\sum_{v_i \in V_s(sg_1) \cup V_s(sg_2)} w(v_i, f)}. \tag{4.8}$$

Therefore, the similarity of the two subgraphs presented in Fig. 4.4 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 examples also demonstrate that our subgraph similarity calculation approach can well tolerate minor differences of implementation.

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 similarity between sg_1 and sg_2 is not higher than $\frac{min(w(sg_1,f),w(sg_2,f))}{max(w(sg_1,f),w(sg_2,f))}$, which can be used to reduce the number of pair-wise graph matching in subgraph clustering.

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

Algorithm 3 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, sg_i is added in c', otherwise a new cluster that contains only sg_i is created and added in C.

ϵ is an important parameter in Algorithm 3. 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.

Definition 2 ***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.*

4.2.3 Construction of Features

To enable malware familial analysis, all fregraphs in the 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.

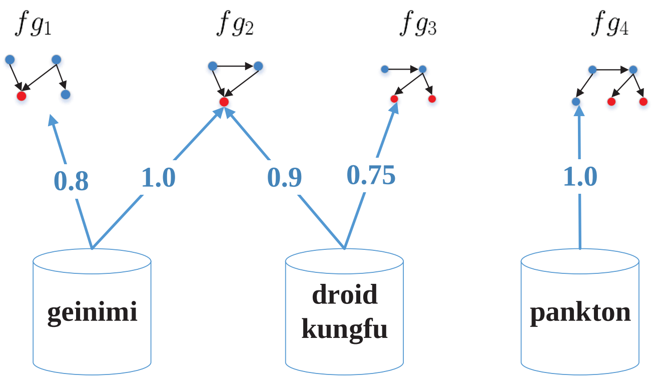


Figure 4.5: A mapping between four fregraphs and three malware families

Mapping exists between fregraphs and families given that some fregraphs belong to more than one family. Fig. 4.5 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:

$$fs(fg) = cb'(fg) * \sum_{f_j \in F} w(fg, f_j) * p(f_j|fg), \tag{4.9}$$

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. (4.10) as follows:

$$p(f_j|fg) = \frac{sup_{f_j}(fg)}{\sum_{f_i \in F} sup_{f_i}(fg)}. \tag{4.10}$$

$cb'(fg)$ indicates the normalized entropy value of fg. $cb'(fg)$ is obtained through Eqs. (4.11)-(4.12), 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. If $cb'(fg) = 1$, then the fregraph belongs to only one family (e.g., fg_1, fg_3 and fg_4 in Fig. 4.5).

$$cb(fg) = \sum_{f_j \in F} p(f_j|fg) * \log p(f_j|fg) \tag{4.11}$$

$$cb'(fg) = \frac{cb(fg) - cb_{min}}{cb_{max} - cb_{min}} \tag{4.12}$$

4.3 Usages of Fregraphs

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

4.3.1 Familial Identification of Android Malware

FalDroid initially constructs a fregraph-based feature vector to represent each malware sample. Within the vector, the default value of each fregraph-based feature is 0, and it will be set to the weighted score when a malware sample contains this feature. For known malware 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.

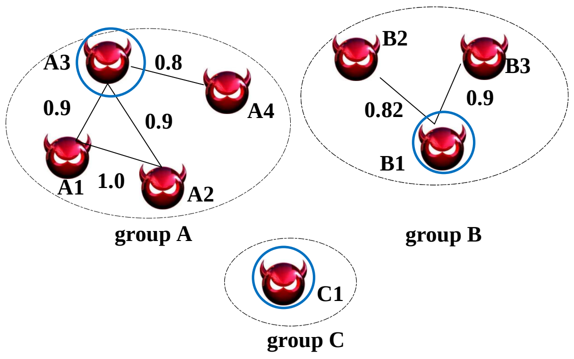


Figure 4.6: An example of MSG

4.3.2 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 3 ***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. 4.6 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 $\overrightarrow{w}$; $|\overrightarrow{u}| = |\overrightarrow{w}| = l$.

$$sim(\alpha_1, \alpha_2) = \frac{\overrightarrow{u} \cdot \overrightarrow{w}}{\|\overrightarrow{u}\|\|\overrightarrow{w}\|} = \frac{\sum_{i=1}^{l} \overrightarrow{u_i}\overrightarrow{w_i}}{\sqrt{\sum_{i=1}^{l} \overrightarrow{u_i}^2}\sqrt{\sum_{i=1}^{l} \overrightarrow{w_i}^2}} \tag{4.13}$$

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

$$\alpha' = \underset{\alpha \in GV(group)}{argmax} \sum_{\beta \in SN(\alpha)} sim(\alpha, \beta), \tag{4.14}$$

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. 4.6, the representative malware samples include A3, B1, and C1, which are marked with blue circles. There may be groups which contain only one sample, such as group C. 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.

Table 4.2: Descriptions of four different datasets

Dataset	#Samples	#Families	Average Size (MB)	Time
Genome Project dataset [156]	1,247	33	1.3	2011∼2012
Drebin dataset [34]	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

4.4 Evaluation of FalDroid

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

RQ 1 *Can FalDroid classify the new malware sample into its family with high accuracy? (Section 4.4.2)*

RQ 2 *Can FalDroid effectively decrease the number of malware samples to be analyzed? (Section 4.4.3)*

RQ 3 *Can FalDroid work efficiently and be scalable for a large number of apps? (Section 4.4.4)*

RQ 4 *Is FalDroid resilient to polymorphic variants and code obfuscation techniques? (Section 4.4.5)*

4.4.1 Study Setup

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 [34] and Android Malware Genome Project [156]. Table 4.2

Table 4.3: 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

lists the descriptions of the four datasets. Among the samples in 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 [34] contains 5,513 samples in 132 malware families, and the dataset provided from Android Malware Genome Project [156] contains 1,247 samples in 36 families.

To construct the FalDroid-I dataset, approximately 15,000 malware samples are first downloaded from VirusShare [28] and uploaded to VirusTotal [29], 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 [116]. Part of the dictionary is listed in TABLE 4.3. 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 4.4, where Num is the number of malware samples in each family. The samples in the FalDroid-II dataset are provided by contagion [25] and MassVet [47] and labeled in the same manner as those in

Table 4.4: 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

the FalDroid-I dataset. Finally, 643 malware samples in 43 families are labeled. Table 4.5 lists the metrics used to evaluate FalDroid.

4.4.2 Accuracy of Familial Identification

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) [45], Decision Tree (C4.5) [120], k-nearest neighbor (k-NN; k=1) [32] and Random Forest (tree num=100) [43]. The experiment is conducted using 10-fold cross-validation.

Fig. 4.7 shows the classification accuracies of the four classifiers with different support

Table 4.5: 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.

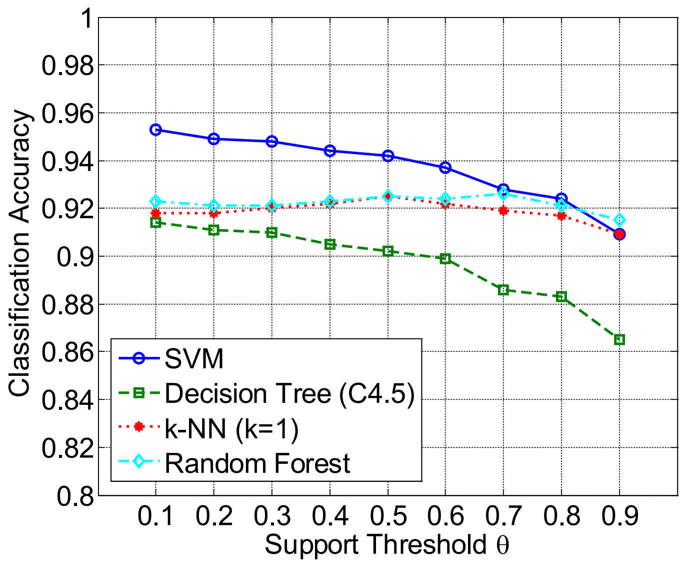


Figure 4.7: Classification performance of FalDroid for four different classifiers under different support thresholds θ

threshold θ ranging from 0.1 to 0.9. We can draw the following three conclusions from Fig. 4.7:

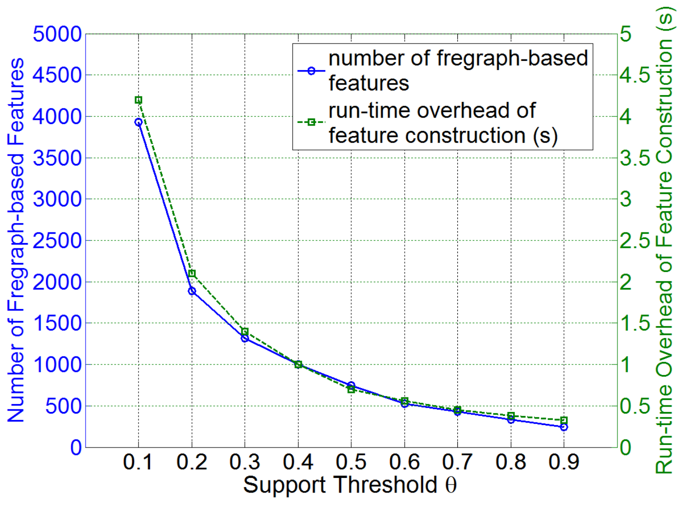


Figure 4.8: Number of fregraph-based features and corresponding run-time overhead of feature construction under different support thresholds θ

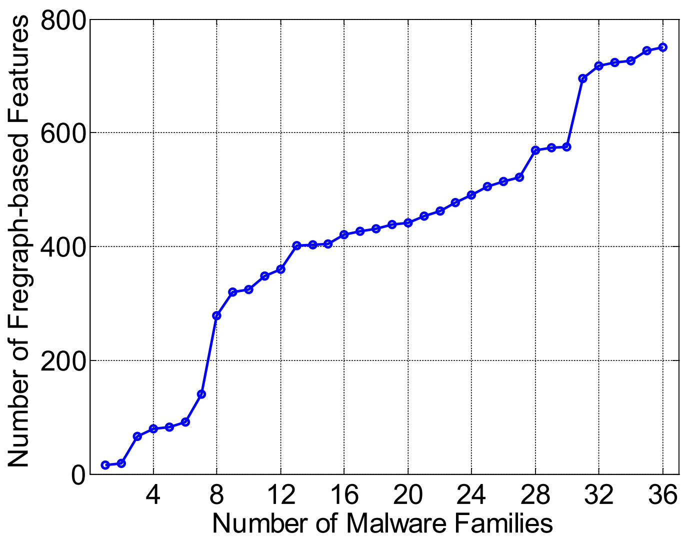


Figure 4.9: Number of fregraph-based features by adding families when $\theta = 0.5$

(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. 4.8, 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. 4.8 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 the number of features decreases by 82% and the run-time overhead of feature construction decreases by 82% when $\theta = 0.1$. Thus, we select SVM as our classifier and set $\theta = 0.5$ in latter experiments. Fig. 4.9 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 4.6 shows the classification results when $\theta = 0.5$. Most families have TPR higher than 0.9. Specifically, 12 families achieve TPR equal to 1 with FPR equal to 0, indicating that all of their samples are accurately classified and no other malware 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 [156]. In summary, FalDroid performs effectively for most families.

2) Classification Performance on Different Datasets: We also evaluate FalDroid using four different datasets. Table 4.7 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 [34] into their families. Its classification accuracy is 0.972 for the Genome Project dataset [156]. Misclassifications are attributed to two main reasons: First, few fregraphs are generated for some families, such as *boxer* in [34], thus causing performance to deteriorate. Second, some families, such as *DroidKungFu2* and *DroidKungFu3* in [156], exhibit similar

Table 4.6: 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 4.7: Classification performance on four datasets

Dataset	Classification Accuracy
Genome Project dataset [156]	0.972
Drebin dataset[34]	0.953
FalDroid-I dataset	0.942
FalDroid-II dataset	0.919

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 [127], Apposcopy [61], DroidSIFT [149], MudFlow [38], TriFlow [99], DroidLegacy [54], and Astroid [62]. These approaches are briefly described below:

- Dendroid automatically classifies malware and analyzes families on the basis of code structures [127].
- Apposcopy extracts the data-flow and control-flow properties of an app to identify its family [61].
- DroidSIFT is a semantic-based approach that classifies malware via API dependency graphs [149].
- MudFlow [38] and TriFlow [99] analyze malware samples on the basis of the source-and-sink method pairs extracted by FlowDroid [35].
- DroidLegacy partitions the app code into loosely coupled modules and identifies the malicious module of each piggybacked malware family [54].

Table 4.8: Classification accuracies of FalDroid and seven state-of-the-art approaches on Genome Project dataset [156].

Baseline Approach	Classification Accuracy
Dendroid [127]	0.942
Apposcopy [61]	0.900
DroidSIFT [149]	0.930
MudFlow [38], TriFlow [99]	0.881
DroidLegacy [54]	0.929
Astroid [62]	0.938
FalDroid	**0.972**

- Astroid automatically synthesizes a maximally suspicious common subgraph of each malware family as a signature to perform familial classification [62].

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 [156], which has been used to evaluate these systems in their works. TABLE 4.8 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.

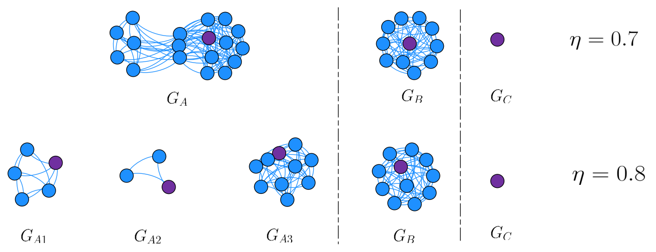


Figure 4.10: MSGs of *zitmo* with $\eta = 0.7$ and $\eta = 0.8$

4.4.3 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. 4.10 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 4.9 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 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 4.10 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

Table 4.9: 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 4.10: The selection of representative malware samples with different similarity threshold values η

Malware Famlily	#groups				Malware Famlily	#groups			
(#samples)	$\eta = 0.5$	$\eta = 0.6$	$\eta = 0.7$	$\eta = 0.8$	(#samples)	$\eta = 0.5$	$\eta = 0.6$	$\eta = 0.7$	$\eta = 0.8$
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

when η increases because it is more difficult for two nodes in MSG to have one edge. We draw the following three conclusions from TABLE 4.10.

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

4.4.4 Analysis of Run-time Overhead

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. 4.12 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 that of the original FCG.

Then, we divide the SARG into a set of sensitive subgraphs using community detection algorithms. Fig. 4.13 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. 4.13 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

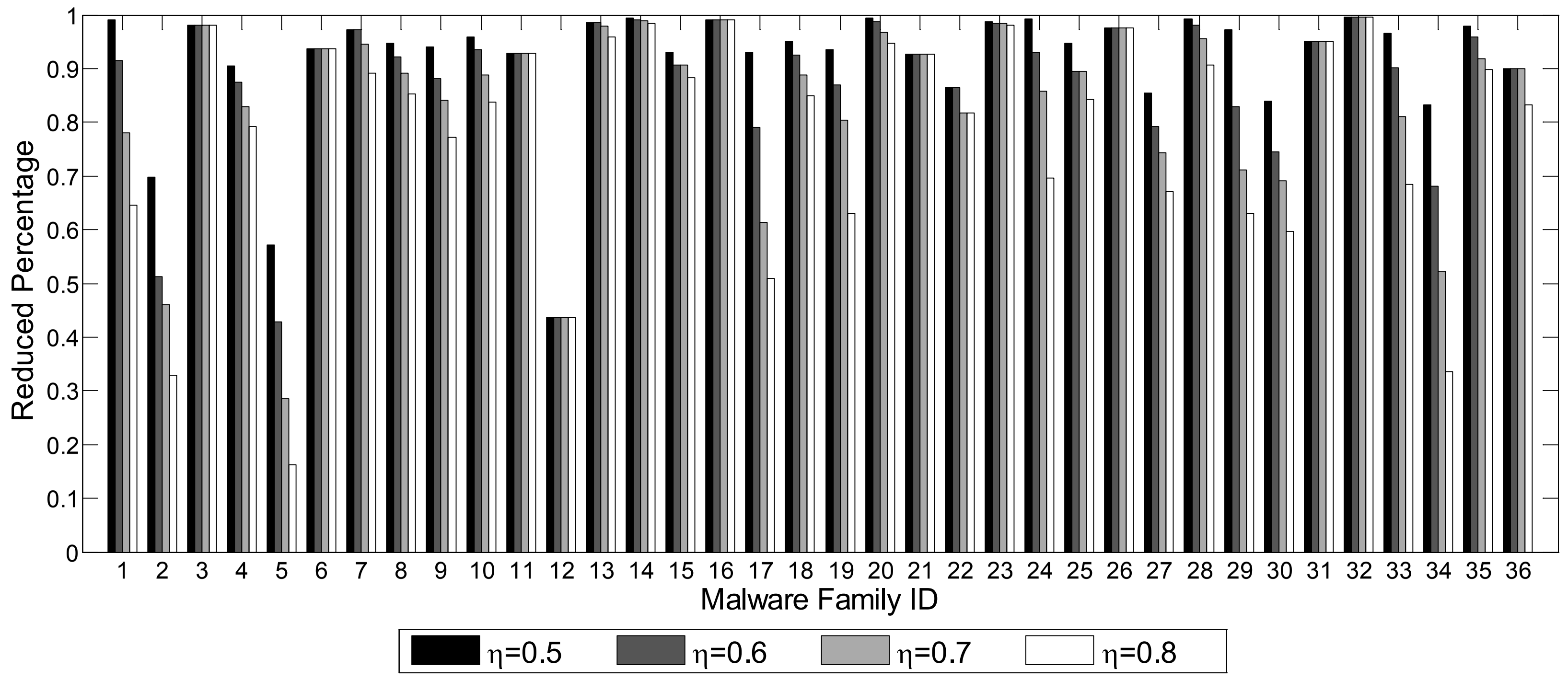


Figure 4.11: Reduced percentages of samples to be analyzed in each family with different similarity threshold values η

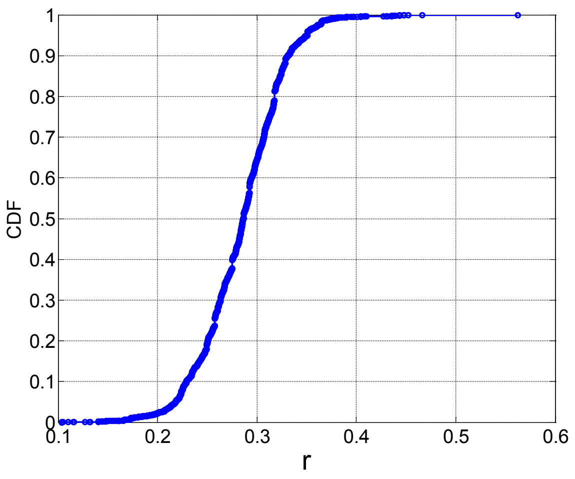


Figure 4.12: CDF of the ratio of size of SARG to its FCG

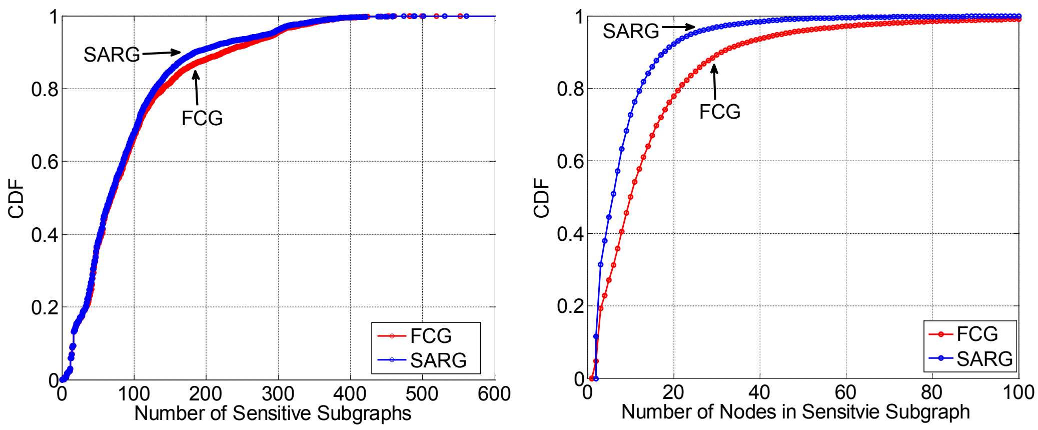


Figure 4.13: CDFs of the number of sensitive subgraphs and the number of nodes in sensitive subgraphs

because the run-time performance of graph matching depends on the number of sensitive subgraphs and their nodes.

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

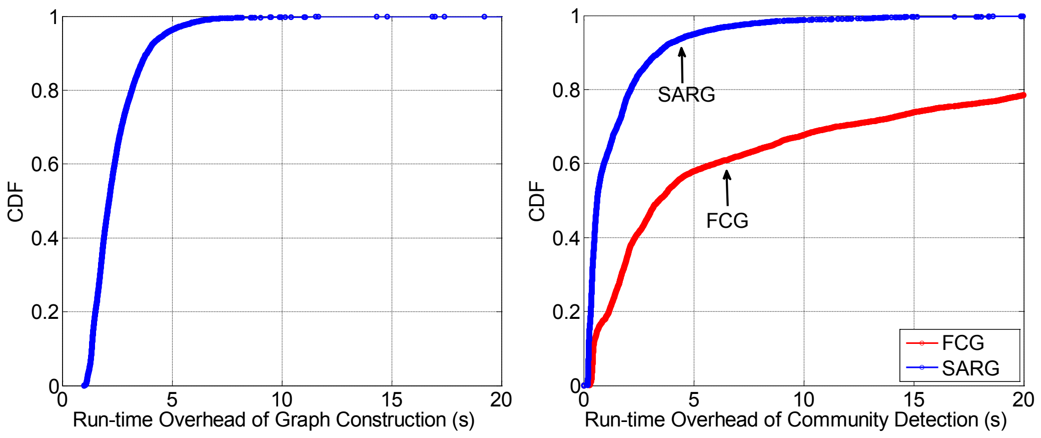


Figure 4.14: CDFs of run-time overhead for graph construction and community detection

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. 4.14. 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 [149] and Apposcopy [61], 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.

In addition, our SVM classifier requires less than 30 sec to completely process the training and testing datasets.

4.4.5 Analysis of Resilience

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 [149, 82]. The GED metric 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 because the node label can be easily changed by obfuscation techniques. 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.

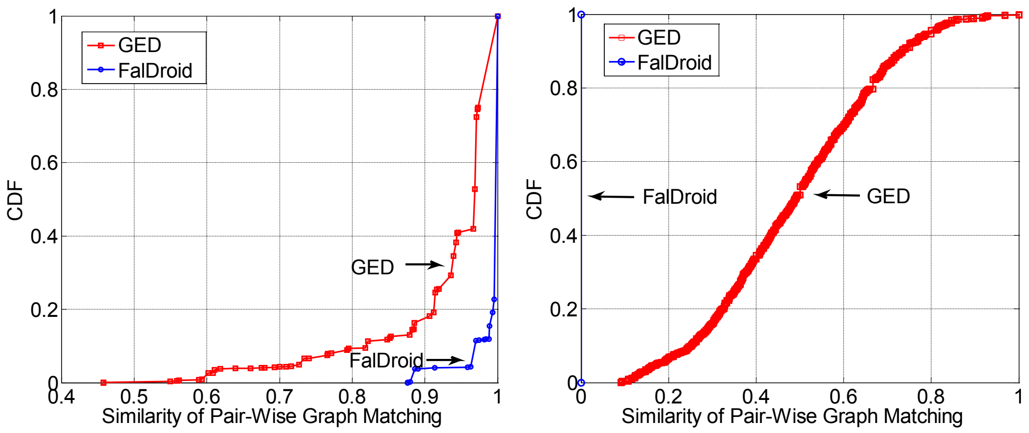


Figure 4.15: Comparison between our weighted-sensitive-API-call-based graph matching approach with GED on the *similar set* and the *dissimilar set*.

For the two subgraph sets, each subgraph is matched with other subgraphs. Therefore, 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. 4.15) and the *dissimilar set* (illustrated in the right-hand side of Fig. 4.15). 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.

Since FalDroid is also based on the sensitive subgraph analysis, which is similar to the proposed DAPASA in Chapter 3, the resilience of FalDroid to other obfuscation techniques (i.e., renaming, reflection, control flow obfuscation) is similar as DAPASA.

4.5 Brief Summary

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.

Chapter 5

The Construction of Graph Embedding Feature and Its Application in Familial Clustering

5.1 Overview

Although Chapter 4 proposes an effective familial identification approach, it still suffers from two main limitations: 1) Low efficiency, even graph-based features such as fregraph could profile the behaviors of malware samples, the similarity calculation of the fregraphs is bounded by the efficiency of existing graph matching approaches [115]. For the proposed weighted-sensitive-API-call-based graph matching approach, it still requires a lot time while handling millions of graphs. 2) Lack of labeled dataset, it is time-consuming and labor-intensive to label a large scale of malware samples with family names. Moreover, since classifiers are trained using known malware samples, they cannot correctly classify new malware samples from unknown families. Note that retraining the classifier model for every new sample may be impractical [143].

To tackle these challenges, we propose GefDroid, a novel **G**raph **e**mbedding based

familial analysis approach of An**Droid** malware with the following salient features:

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

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

In summary, our major contributions include:

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

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

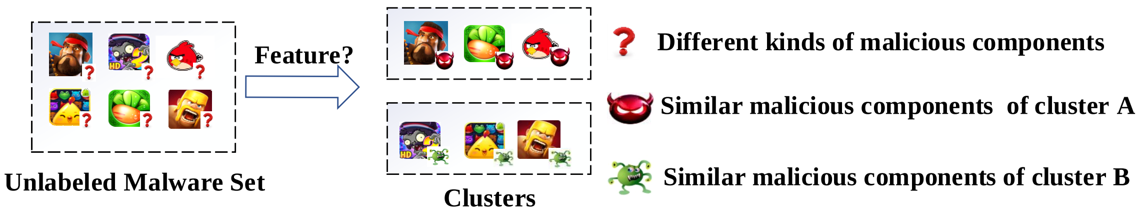
(iii) We conduct extensive experiments to evaluate GefDroid. The results show that GefDroid can achieve high agreements (0.707-0.883 in term of NMI) between our clustering results and the ground truth datasets. Furthermore, GefDroid requires only linear run-time overhead and takes around 8.6s to analyze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

The rest of this chapter is organized as follows. Section 5.2 introduces the problem. Section 5.3 details the construction of graph embedding feature and Section 5.4 introduces the application of the graph embedding feature in familial analysis. Section 5.5 reports the experimental results. Section 5.6 concludes the chapter.

5.2 Motivation and Problem Definition

5.2.1 Motivating Scenario

Let us consider a security analyst who faces thousands of unlabeled malware samples captured every day as illustrated in Fig. 5.1. These malware samples are generally produced by injecting different kinds of malicious components into popular apps. The analyst aims to find and analyze the new injected malicious components. However, it is time-consuming and labor-intensive to conduct an in-depth analysis on each sample. Therefore, the analyst tries to group these malware samples into a set of clusters, where the samples belonging to the same cluster share similar malicious components. By inspecting the similar malicious components in each cluster, the analytical workload of the analyst can be effectively reduced. However, the analyst faces two challenges: First, how to effectively identify the malicious components that usually constitute only a small portion of the samples and may not be implemented in the same way? Second, how to efficiently accomplish the clustering of thousands of malware samples with low overhead? Note that



Feature?
Unlabeled Malware Set
Clusters
Different kinds of malicious components
Similar malicious components of cluster A
Similar malicious components of cluster B

Figure 5.1: Motivation scenario of GefDroid

directly applying pair-wise exact matching is neither effective nor efficient.

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

5.2.2 Problem Definition

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

$$M_{fea} \times M_{fea} \Rightarrow L \tag{5.1}$$

Then, the constructed MLN is similar to a social network, where each malware sample is regarded as an entity and each edge is regarded as the relationship that connects entities. Moreover, the malware families that we aim to find are regarded as the communities existed in the network. In general, community detection algorithms are widely used to detect community structures in social networks, thus they can be applied on our constructed MLN in a similar way. Formally, after constructing the MLN, we aim to find the families as:

$$\mathcal{C}(MLN) \Rightarrow Y = \{y_1, y_2, \ldots, y_R\} \tag{5.2}$$

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

5.3 Construction of Graph Embedding Feature

5.3.1 Graph Embedding

After the construction of sensitive subgraphs, it is straightforward to apply graph matching algorithms (e.g., bipartite graph matching [115]) to perform app similarity detection like FalDroid does. However, the graph matching algorithms are slow since they require super-linear time running in the graph size. Furthermore, similarity among hundreds of thousands of graphs must be calculated. Thus, the approaches [149, 51, 52] based on graph matching algorithms are inevitably inefficient.

In recent years, deep learning [84] has been applied to many application domains, including graph embedding [103, 105, 72, 113, 53], which aims at learning low-dimensional vector representations for nodes of a given graph. Graph embedding has been proven to be useful in many tasks of graph analysis, including link prediction [89], node classification [40], and visualization [93]. The learned low-dimensional vector representations for nodes can effectively transform the high-cost graph matching to an easy-to-compute distance calculation between vectors.

In our approach, the applied graph embedding technique should satisfy two requirements. First, given that new malware samples are constantly being discovered, the graph embedding algorithm should work with the input of only one graph per time rather than a graph set. In this way, the trained model does not need retraining process for the new coming samples. Second, the latent representation of nodes should not depend on the node or edge attribute, especially the node labels (i.e., method names) that can be easily changed by obfuscation techniques. Consider integrating the performance and scalability, we use *struc2vec* [113] as our default graph embedding technique.

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

Fig. 5.2 presents an example of an undirected graph that contains 11 nodes and 11 edges. The embedding results of the example graph are illustrated in Fig. 5.3, where the dimension argument is set as 2 for visualization here. As can be seen from the two figures, the learned feature vectors of the same nodes are quite different due to the random walk strategy used in *struc2vec*. Thus, it is not effective to directly apply the embedding technique in our work.

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

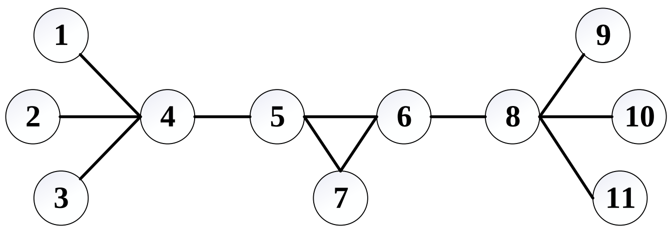


Figure 5.2: An example of an undirected graph that contains 11 nodes and 11 edges, where node 4 and node 8 are structurally similar since both of their degrees are 4.

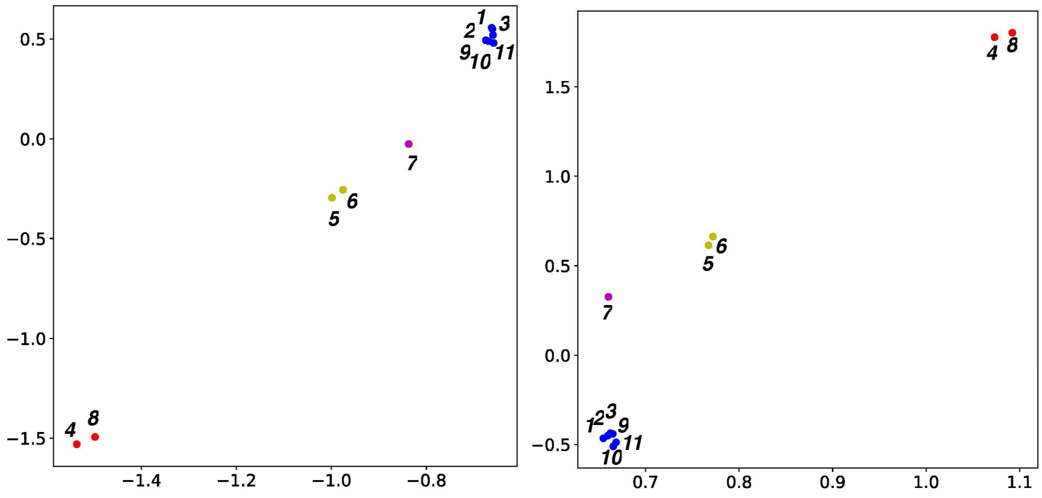


Figure 5.3: Visualization of the embedding results of the same graph after twice applying of *struc2vec* with the same arguments.

5.3.2 SRA Generation

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

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

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

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

$$h_t(v, u) = \cos(\mathbf{u}_v, \mathbf{u}_u) = \frac{\mathbf{u}_v \cdot \mathbf{u}_u}{\|\mathbf{u}_v\| \|\mathbf{u}_u\|} \tag{5.3}$$

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

Finally, SRA_t is obtained as:

$$SRA_t = \{h_t(v, u) | v, u \in SS_t \quad and \quad v \neq u\}. \tag{5.4}$$

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

5.3.3 SRA Similarity Calculation

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

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

On the basis of the above two intuitions, the similarity of two given SRAs generated from sg_1 and sg_2, denoted as $sim(SRA_1, SRA_2)$, is obtained with Eq. (5.5).

$$sim(SRA_1, SRA_2) = \frac{\sum_{v_i \in SS_1 \cap SS_2} sim(\mathbf{sr}_1(v_i), \mathbf{sr}_2(v_i))}{|SS_1 \cup SS_2|} \tag{5.5}$$

where $\mathbf{sr}_1(v_i)$ and $\mathbf{sr}_2(v_i)$ are represented as two vectors and they denote the similarity relationships between node v_i with other sensitive API call nodes in subgraphs sg_1 and sg_2, respectively. To obtain $\mathbf{sr}_1(v_i)$ and $\mathbf{sr}_2(v_i)$, for convenience, we first construct two distance matrices $D_t(t = 1, 2)$ for two subgraphs as Eq. (5.6). Then $\mathbf{sr}_t(v_i)$ is the i^{th} row vector of the constructed matrices as Eq. (5.7).

$$D_t[i, j] = \begin{cases} h_t(v_i, v_j) & v_i, v_j \in SS_1 \cap SS_2, i \neq j \\ 0 & i = j \end{cases} \tag{5.6}$$

$$\mathbf{sr}_t(v_i) = D_t[i, :] \tag{5.7}$$

$$sim(\mathbf{sr}_1(v_i), \mathbf{sr}_2(v_i))) = \frac{1}{1 + \|\mathbf{sr}_1(v_i) - \mathbf{sr}_2(v_i)\|_2} \tag{5.8}$$

Fig. 5.4 presents an example of the similarity calculation of two SRAs generated from sg_1 and sg_2. Note that only parts of the subgraphs are shown, the other parts located in rectangles are quite different. The two subgraphs have three common sensitive API call nodes (i.e., red nodes v_1, v_2, and v_3). For subgraph sg_1, $h_1(v_1, v_2) = 0.4$

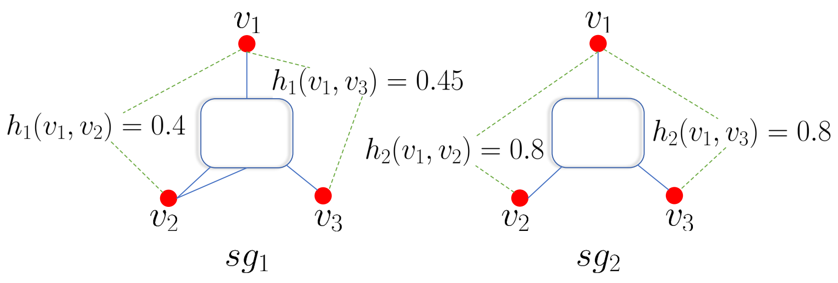


Figure 5.4: An example of the similarity calculation of two SRAs generated from sg_1 and sg_2.

and $h_1(v_1, v_3) = 0.45$. For subgraph sg_2, $h_2(v_1, v_2) = h_2(v_1, v_3) = 0.8$. Therefore, $sim(\mathbf{sr}_1(v_1), \mathbf{sr}_2(v_1))) = sim(< 0.4, 0.45 >, < 0.8, 0.8 >)$. Note that the cosine metric result of the two vectors is 0.998. However, the high similarity calculated with cosine metric cannot depict the different sensitive API invocation patterns here. Thus, we apply the Euclidean metric with Eq. (5.8) rather than the cosine metric. The Euclidean metric result is 0.653, considerably less than the result of cosine metric.

5.4 Familial Clustering

5.4.1 MLN Construction

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

Algorithm 4 lists the steps of constructing the MLN with the input of malware set M and two threshold values, θ and ϵ. θ denotes the similarity threshold value between SRAs.

In other words, if the similarity of two SRAs calculated with Eqs. (5.5-5.8) is no less than θ, they are regarded as the same, indicating that their corresponding classes share similar functionalities. ϵ denotes the threshold value of adding edges between sample nodes. If the number of same SRAs shared by two samples is no less than ϵ, then an edge is added between the two samples.

In Algorithm 4, after the preprocessing of each sample in M (lines 2), a set of SRAs, denoted as $SRASet$, is constructed (lines 3-6). Then each sample is added in the MLN as a node (line 7). After that, for each sample-pair in M, the number of same SRAs between them is calculated and represented as k (lines 10-17). An edge with weight k is added for the sample-pair if there exist no less than ϵ same SRAs between them (lines 18-20).

Algorithm 4 Construction of MLN

```
Require: $M$   // $M$ denotes the set of malware samples to be analyzed.
         $\theta$   // $\theta$ denotes the similarity threshold value between $SRA$s.
         $\epsilon$   // $\epsilon$ denotes the threshold value of adding edges between nodes.
Ensure: $MLN = \{M, L\}$
 1: for each malware sample $m_i$ in $M$ do
 2:    $SGS_{m_i} = \mathcal{F}_{div}(SARG_{m_i})$
 3:    for each $sg_t$ in $SGS_{m_i}$ do
 4:       $SRA_t = GenerateSRA(sg_t)$
 5:    end for
 6:    $SRASet_{m_i} = \{SRA_t | 1 \leq t \leq T\}$
 7:    $MLN.addNode(m_i)$
 8: end for
 9: for each sample-pair $(m_i, m_j)$ in $M$ do
10:    $k = 0$
11:    for each $SRA_t$ in $SRASet_{m_i}$ do
12:       for each $SRA_{t'}$ in $SRASet_{m_j}$ do
13:          if $sim(SRA_t, SRA_{t'}) \geq \theta$ then
14:             $k = k + 1$
15:          end if
16:       end for
17:    end for
18:    if $k \geq \epsilon$ then
19:       $MLN.addEdge(m_i, m_j, k)$ // $k$ denotes the edge weight.
20:    end if
21: end for
22: return $MLN$
```

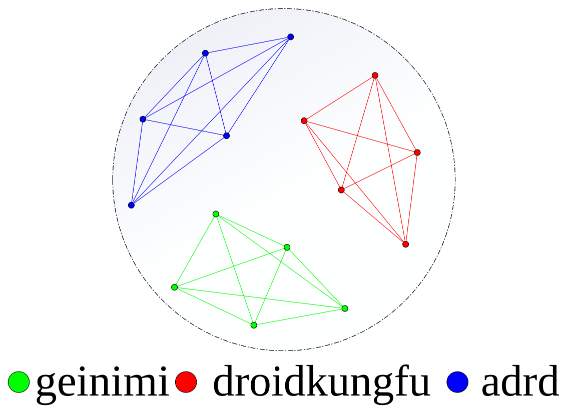


Figure 5.5: An example of community detection result of MLN for fifteen malware samples in three families.

5.4.2 Community Detection on MLN

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

Fig. 5.5 presents an example of community detection result of MLN for fifteen malware samples in three families, i.e., *geinimi*, *droidkungfu*, and *adrd*. It is obvious that the constructed MLN can be divided into three clusters. In each cluster, the samples are connected with each other, indicating that the samples within the same cluster share similar malicious components. On the basis of the clustering results, our approach can effectively help security analysts focus on the commonalities among malware samples within the same cluster, and potentially isolate the malicious behaviors of malware samples from different clusters.

5.5 Evaluation of GefDroid

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

RQ 1 *Which community detection algorithm is appropriate for GefDroid? (Section 5.5.2)*

RQ 2 *Does GefDroid outperform the baseline approaches in term of accuracy? (Section 5.5.3)*

RQ 3 *Can GefDroid process a great deal of samples with low run-time overhead? (Section 5.5.4)*

RQ 4 *How do the two parameters influence the performance of GefDroid? (Section 5.5.5)*

5.5.1 Study Setup

We evaluate GefDroid on three ground truth datasets provided by Genome project [156], Drebin [34], and Fan *et al.* [59]. For convenience, they are named as dataset-I, dataset-II, and dataset-III. We use Q and K to denote the number of families and the number of malware samples, respectively. Thus, $Q_1 = 49, K_1 = 1,260, Q_2 = 179, K_2 = 5,560, Q_3 = 36, K_3 = 8,407$.

Six metrics are used to measure the clustering performance. They are normalized mutual information (NMI) [57], adjusted rand index (ARI) [126], Fowlkes-Mallows index (FMI) [63], Homogeneity [117], Completeness [117], and V-measure [117]. NMI, ARI, and FMI are three widely-used metrics that measure the agreement between the clustering result and the ground truth dataset. Homogeneity measures the extent of how each generated cluster contains only samples of a single family. Completeness measures the extent of how all samples of each family are assigned to the same cluster. V-measure is the harmonic mean of homogeneity and completeness.

Except for the ARI, the values of the other five metrics range from 0 to 1, where a higher value indicates a better agreement between the predicted clusterings and the true clusterings. The value of ARI ranges from -1 to 1, where random labelings have an ARI value close to 0.0. For all the six metrics, 1.0 stands for a perfect match with the ground truth dataset. Recall the example of community detection result illustrated in Fig. 5.5, all the six metrics are 1.0.

We use $Y = \{y_1, y_2, \ldots, y_R\}$ and $C = \{c_1, c_2, \ldots, c_Q\}$ to denote the predicted set of clusters and the true set of families, respectively. R denotes the number of clusters generated with the community detection algorithm; Q denotes the number of malware families. Before calculating the six metrics, we introduce the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) used in familial clustering.

- TP: the number of pairs of samples in the same families in Y and in the same clusters in C.
- TN: the number of pairs of samples not in the same families in Y and not in the same clusters in C.
- FP: the number of pairs of samples not in the same families in Y but in the same clusters in C.
- FN: the number of pairs of samples in the same families in Y but not in the same clusters in C.

Then the calculation of the six metrics are listed as follows.

NMI: NMI is the normalized mutual information that measures the agreement between two partitions in clustering analysis, the given Y and C. It is calculated with Eq. (5.9).

$$NMI(Y, C) = \frac{MI(Y; C)}{\sqrt{H(Y)H(C)}} \tag{5.9}$$

$MI(Y; C)$ denotes the mutual information between Y and C. It is calculated with Eq. (5.10):

$$MI(Y; C) = \sum_{r=1}^{R} \sum_{q=1}^{Q} P(y_r \cap c_q) \log \frac{P(y_r \cap c_q)}{P(y_r)P(c_q)}, \tag{5.10}$$

where $P(y_r)$, $P(c_q)$, and $P(y_r \cap c_q)$ are the probabilities of a malware sample being in cluster y_r, c_q, and the intersection of y_r and c_q, respectively. $H(Y)$ and $H(C)$ are the entropies calculated with Eq. (5.11) and Eq. (5.12).

$$H(Y) = -\sum_{r=1}^{R} P(y_r) \log P(y_r) \tag{5.11}$$

$$H(C) = -\sum_{q=1}^{Q} P(c_q) \log P(c_q) \tag{5.12}$$

ARI: ARI is also a metric that measures the similarity of two two partitions in clustering analysis. It is calculated on the basis of rand index (RI).

$$RI = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.13}$$

However the RI score does not guarantee that random label assignments will get a value close to zero. To counter this effect, ARI is calculated as:

$$ARI = \frac{RI - E[RI]}{\max(RI) - E[RI]}, \tag{5.14}$$

where $\max(RI)$ and $E[RI]$ denote the maximum RI score and the expected RI score, respectively.

FMI: It is another metric to measure the accuracy of clustering result. It is calculated as:

$$FMI = \frac{TP}{\sqrt{(TP+FP)(TP+FN)}}. \tag{5.15}$$

Homogeneity: Homogeneity is used to measure the extent of how each generated cluster contains only malware samples of a single family. It is calculated as:

$$homogeneity = 1 - \frac{H(C|Y)}{H(C)}, \tag{5.16}$$

where $H(C|Y)$ is the conditional entropy of the classes given the clustering result and is given by Eq. (5.17). $H(C)$ is calculated with Eq. (5.12).

$$H(C|Y) = -\sum_{q=1}^{Q}\sum_{r=1}^{R} P(y_r \cap c_q)\log(\frac{P(y_r \cap c_q)}{P(y_r)}) \tag{5.17}$$

Completeness: Completeness is used to measure the extent of how all malware samples of a given family are assigned to the same cluster. It is calculated as:

$$completeness = 1 - \frac{H(Y|C)}{H(Y)}. \tag{5.18}$$

$H(Y|C)$ and $H(Y)$ are calculated in a symmetric manner as $H(C|Y)$ and $H(C)$, respectively.

V-measure: V-measure is the harmonic mean of homogeneity and completeness and it is calculated as:

$$V - measure = 2 * \frac{homogeneity * completeness}{homogeneity + completeness}. \tag{5.19}$$

Table 5.1: Clustering performance of GefDroid with four different community detection algorithms on three datasets.

Dataset	Community detection algorithm	NMI	ARI	FMI	Homogeneity	Completeness	V-measure	#clusters
dataset-I (Q=49, K=1,260)	Infomap	**0.883**	**0.870**	**0.886**	**0.892**	0.876	**0.883**	74
	Fast greedy	0.701	0.602	0.661	0.632	0.778	0.698	72
	Label propagation	0.859	0.827	0.852	0.839	**0.880**	0.859	60
	Multilevel	0.795	0.765	0.804	0.729	0.866	0.792	61
dataset-II (Q=179, K=5,560)	Infomap	**0.793**	**0.534**	**0.606**	**0.924**	**0.681**	**0.784**	471
	Fast greedy	0.713	0.473	0.513	0.764	0.665	0.711	464
	Label propagation	0.757	0.434	0.497	0.879	0.653	0.749	409
	Multilevel	0.738	0.513	0.557	0.809	0.674	0.735	409
dataset-III (Q=36, K=8,407)	Infomap	**0.707**	**0.509**	**0.559**	**0.832**	0.600	**0.697**	685
	Fast greedy	0.592	0.340	0.405	0.614	0.571	0.592	659
	Label propagation	0.671	0.426	0.470	0.757	0.594	0.666	619
	Multilevel	0.652	0.452	0.495	0.706	**0.601**	0.650	619

5.5.2 Selection of Community Detection Algorithm

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

- Infomap, which detects community structures of a network using the approach proposed by Rosvall *et al.* [118].
- Fast greedy, which is based on the greedy optimization of modularity [49], which is a metric to measure the quality or significance of a community structure.
- Label propagation, which is a fast partitioning algorithm proposed by Raghavan *et al.* [109].
- Multilevel, which is a layered and bottom-up community detection algorithm proposed by Blondel *et al.* [42].

In our experiment, the default values of the two arguments, θ and ϵ, are set as 0.75 and 1, respectively. The sensitivity of the arguments to the clustering performance will be discussed in Section 5.5.5.

Table 5.1 lists the clustering performance with four different community detection algorithms on three datasets. The term #clusters denotes the number of generated clusters. The values marked in bold denote the best performance in terms of the six metrics among the four algorithms. According to the results listed in Table 5.1, the infomap algorithm achieves the best clustering performance among these four algorithms. The NMI values of infomap on the three datasets are higher than 0.7, indicating that there exists high agreements between the clustering results and the ground truth datasets. Thus, infomap is selected as our default community detection algorithm in the latter experiments.

5.5.3 Accuracy of Malware Familial Clustering

We compare the accuracy of GefDroid with four baseline approaches that are briefly introduced as below:

- Wang *et al.* proposed an approach for malware detection based on the requested permissions, which are security-aware features that restrict the access of apps to the core facilities of devices [137].
- Aafer *et al.* proposed an approach for malware detection based on API calls, which are more fine-grained features than permissions since each permission governs several API calls [31].
- We proposed FalDroid in Chapter 4, which performs familial classification based on the generated fregraphs that denote the common behaviors of malware samples within the same families [58].
- Marastoni *et al.* proposed GroupDroid, which uses 3D-CFG centroids [46] as features to measure the similarities between malware samples and perform grouping [94].

Among these approaches, GroupDroid [94] performs a clustering task like GefDroid does, while the other three approaches [137, 31, 58] perform a classification task and they suffer from two main limitations. First, they require a training dataset with family labels assigned by experts, which is not easy to obtain. Second, they can only identify the families that are only provided in the training dataset. Thus, for Android familial analysis, it is more practical to perform a clustering task as we do rather than performing a classification task.

Table 5.2: Clustering performance of GefDroid and four baseline approaches with infomap algorithm on three datasets.

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

To have a fair comparison of clustering performance with the approaches that perform a classification task [58, 137, 31], we construct different MLNs for such approaches based on their proposed features, e.g., fregraphs, permissions, and API calls. Then the infomap algorithm is applied on their MLNs to perform a clustering task. For GroupDroid [94], we re-implement it and perform a clustering task based on the extracted 3D-CFG centroids. In addition, we use GefDroid (w/o NR) to denote our approach without the preprocessing of noise removal, in order to evaluate whether the third-party or advertisement libraries affect the clustering performance.

The comparison results are listed in Table 5.2, where the term #Cluster denotes the number of generated clusters. We can draw the following four conclusions from the results:

(i) Except for the completeness metric, GefDroid performs best among these approaches in terms of the other five metrics.

(ii) GefDroid generates the most clusters. The highest homogeneity values and the most clusters indicate that GefDroid can well isolate the malicious behaviors of malware samples from different families.

(iii) In general, the string-based features (i.e., permissions and API calls) perform worse than the graph-based features. The main reason is that they cannot well depict the program semantic meanings, thus insufficient to mine the common malicious components of malware samples within the same families.

(iv) The preprocessing of noise removal significantly improves the clustering performance, indicating that the widely used third-party or advertisement libraries would introduce noise edges into the MLNs.

Table 5.3: Performance of detecting new coming malware samples.

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

We also evaluate the ability of GefDroid in handling new malware samples. We randomly select 100 samples from each dataset and regard them as new samples. Then, we calculate their similarity relationships with existing samples in the MLN, and use three terms to evalaute the performance.

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

We repeat this experiment 100 times on the three datasets. The average results listed in Table 5.3 indicate that GefDroid can effectively link the new coming samples with their variants in the MLN. Moreover, we find that 0.52% of new samples actually belong to the families that contain only one sample in the datasets, and thus they have no links with other samples.

5.5.4 Analysis of Run-time Overhead

We evaluate the run-time overhead of GefDroid for its three main stages that are listed as below:

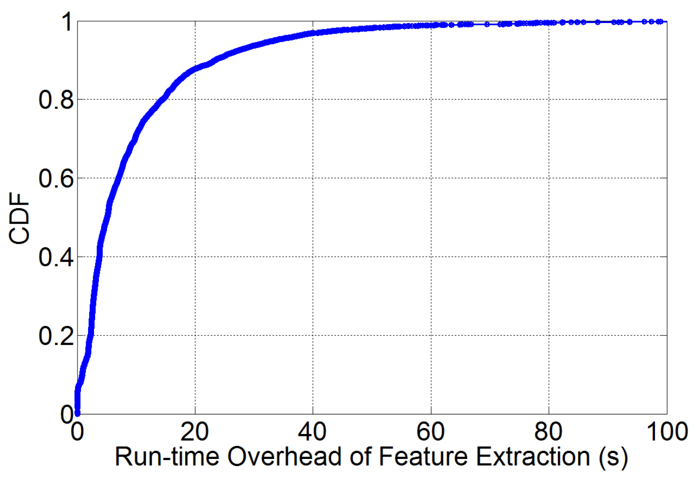


Figure 5.6: CDFs for the run-time overhead of the feature extraction stage on dataset-III.

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

GefDroid-Runtime-Embedding

For the preprocessing stage, 3.9s is needed as introduced in Chapter 4. As illustrated in Fig. 5.6, for the feature extraction stage, 6.5s is needed on average. Furthermore, only about 6.1% of samples require more than 30s. The cost of feature extraction mainly depends on the size of the subgraphs that are embedded. The used embedding algorithm *struc2vec* [113] scales super-linearly but closer to linear. Specifically, the complexity of graph embedding step is $O(n^{1.5})$, where n denotes the number of nodes in a given subgraph. It is worth noting that the preprocessing and the feature extraction stages could

Table 5.4: Run-time overheads of MLN construction and community detection on three datasets.

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

be conducted on several PCs in parallel, thus further reducing the total overhead.

For the stage of familial clustering, an MLN is first constructed by calculating the similarities between SRAs. Thus, the calculation complexity of the SRA pairs is about $O(\frac{K*(K-1)}{2} * \overline{T} * \overline{T})$, where K and $\overline{T}$ denote the total number of samples and the average number of SRAs per sample has, respectively. Table 5.4 lists the run-time overheads of MLN construction and community detection on three datasets. Even for the biggest dataset-III, the similarity calculation of $9.09 * 10^8$ pair of SRAs is accomplished in only 750s. The cost for the community detection is considerably less than that of the MLN construction. Furthermore, for a new coming sample, it only needs 0.2s to calculate the similarities with existing samples.

We compare the overhead of GefDroid and that of the baseline approaches. For the permissions- and API-calls-based approaches, their clustering performance is considerably worse than GefDroid. Moreover, compared with the graph-based features, the permissions- and API-calls-based features cannot provide enough explanations to the relationships between malware samples within the same clusters. Hence, we mainly focus on the efficiency comparison between GefDroid and the two graph-based approaches, i.e., FalDroid and GroupDroid. More precisely, we first randomly select 1,000 malware samples. Then, a set of subgraphs are generated for each sample in the three approaches. After that, FalDroid adopts a weighted-sensitive-API-calls-based graph

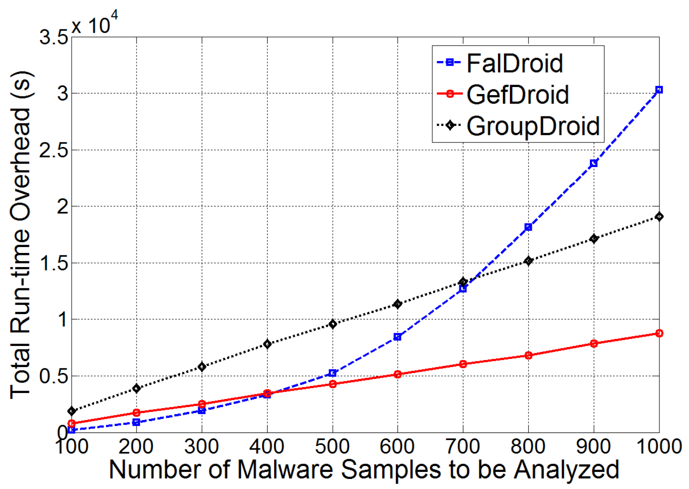


Figure 5.7: Comparison result of the run-time overheads.

matching approach to calculate the graph similarities, which has been proved to be faster than the graph edit distance algorithm. For GroupDroid, a 3D-CFG centroid represented as a four-dimensional vector is calculated for each subgraph. In GefDroid, each subgraph is first embedded into a low-dimensional feature space. Then an SRA is generated for each subgraph. In summary, the similarity detection between samples of FalDroid is based on the graph matching algorithm while GroupDroid and GefDroid rely on similarity calculation between vectors.

Fig. 5.7 use the blue line, black line, and red line denote the increase of total run-time overhead of FalDroid, GroupDroid, and GefDroid, respectively. We can see that the blue line grows exponentially while the black line and the red line show linear growths. About 19s is required for GroupDroid to construct the 3D-CFGs and calculate the centroid, which is twice as the time GefDroid needs. For FalDroid, when the number of samples is lower than 400, it shows higher efficiency than GefDroid by directly applying the graph matching approach. However, with the increase in the number of samples, the cost of FalDroid is considerably higher than GefDroid. For example, when there are 1,000 samples, FalDroid requires around 8.4h to accomplish the similarity calculation of subgraphs, while our approach requires 2.3h to generate the SRAs and accomplishes the similarity calculation between the SRAs under 30s. In reality, there would be much more than 1,000 malware samples to process.

5.5.5 Selection of Arguments

There are two parameters that play importance roles in our approach, i.e., θ controls the similarity threshold value between SRAs; ϵ controls the threshold value of adding edges between sample nodes. To answer RQ 4, we vary the values of θ as $\{0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$ and vary the values of ϵ as $\{1,2,3,4,5\}$ when conducting the experiments on dataset-III. The results are shown in Fig. 5.8. From the six sub figures, we have the following observations:

(i) As the increase of ϵ, except the homogeneity value, the values of the other five metrics decrease.

(ii) As the increase of θ, except the homogeneity value, the values of the other five metrics first increase and then decrease when θ is higher than 0.75.

(iii) For the homogeneity value, it increases with the increase of both θ and ϵ, which makes it harder to connect edges in MLN. However, the homogeneity value can achieve an acceptable 0.832 when $\theta = 0.75$ and $\epsilon = 1$.

We can also draw the similar conclusions from evaluation results on dataset-I and dataset-II. On the basis of the above three conclusions, in our approach, we select the default values of θ and ϵ as 0.75 and 1, respectively.

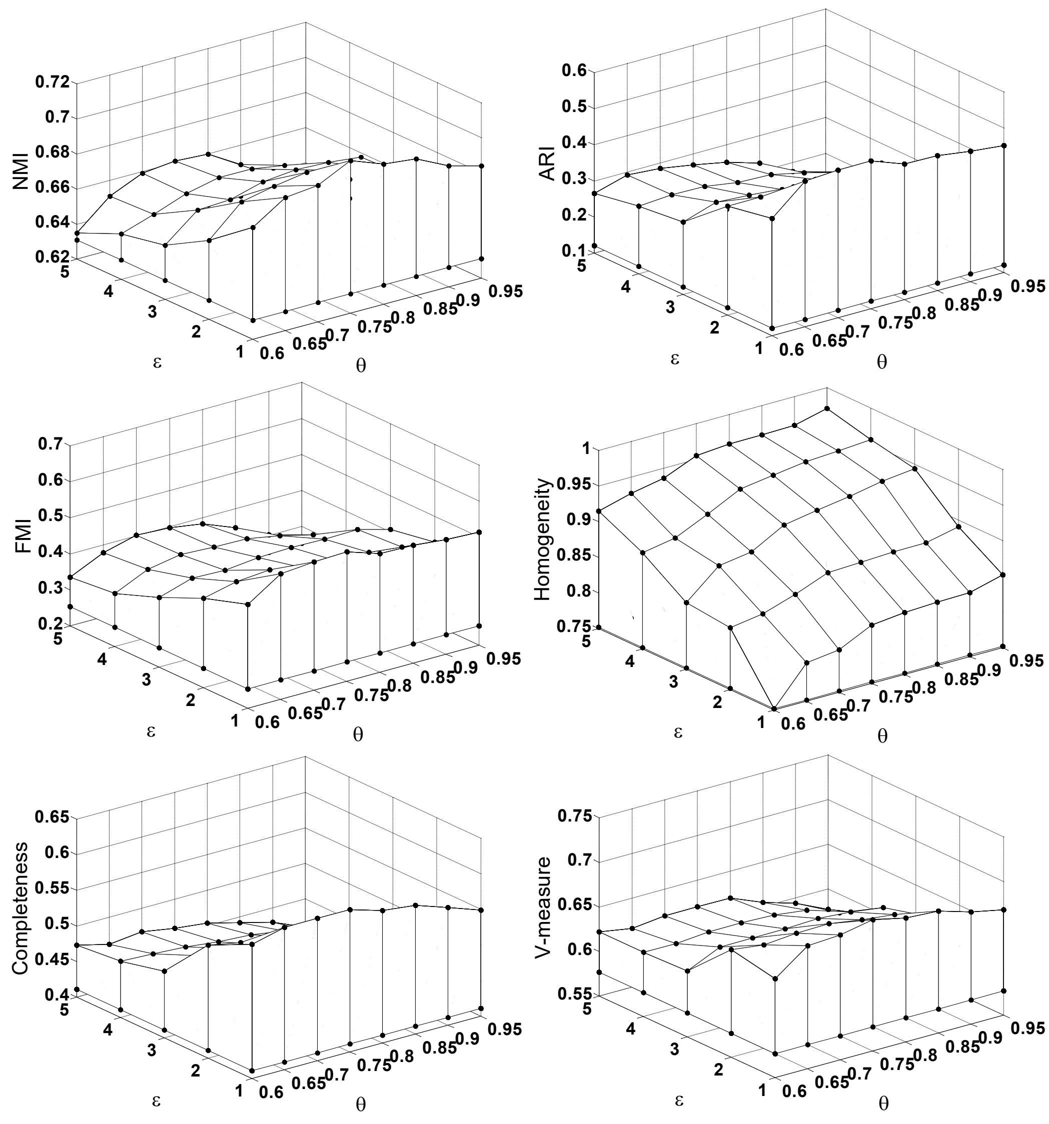


Figure 5.8: Parameter sensitivity of GefDroid for malware familial analysis on dataset-III.

5.6 Brief Summary

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

SRAs. Our extensive evaluation results show that GefDroid outperforms the state-of-the-art approaches in terms of accuracy and efficiency.

Chapter 6

The Construction of Sensitive Behavior Feature and Its Application in Malware Analysis

6.1 Overview

The effectiveness of the above approaches primarily depends on the manual feature engineering process, which is time-consuming and labor-intensive based on human knowledge and intuition. Specifically, to perform malware analysis with high performance, the researchers need to manually inspect the malicious activities of malware samples and summarize the hypotheses about common behaviors that malware share but benign apps do not. Furthermore, the summarized hypotheses might vary from different inspected malware samples, thus constructing different feature spaces for different datasets.

Therefore, in this work, we aim to automatically engineer informative features from existing knowledge learned by experts. Specifically, we mine *sensitive behaviors* features, behaviors that might do harmful activities to users potentially, from a corpus of Android malware related technical blogs. We choose the technical blogs as our knowledge source because they are written in a way that mirrors the human feature engineering process and they are usually public online in time. Then we use the extracted sensitive behaviors to guide the automatic informative feature engineering processing. However, there are two main challenges in our work.

First, it is a key challenge to automatically recognize the harmful activities and mine sensitive behaviors in the magnanimity information of thousands of technical blogs. For example, for the sentence: *"These instructions can be used to open a web page, call a phone number, or send an SMS text message to a premium number.[1]"*, it is easy for the researchers to obtain the knowledge that there are three sensitive behaviors that are marked with an underline for the instructions. However, this conclusion is based on the prior knowledge of research in the world, since the sentence does not provide sufficient linguistic clues that such three behaviors might do harmful activities. Therefore, there is a semantic gap between the natural language in technical blogs and sensitive behaviors.

Second, there also exists a semantic gap between the sensitive behaviors and the programming language. Therefore, it is hard to directly utilize the sensitive behaviors for malware analysis with machine learning algorithms. For example, even though we know that *send an SMS text message* is a sensitive behavior, we still unable to directly identify how does a given app perform such sensitive behavior in their thousands of lines of code.

To overcome the first challenge, we leverage natural language processing (NLP) techniques to parse the contents in blogs into a uniform structure, verb-object phrase (e.g., "send—>text message"). Then we propose a clustering-based approach to extract frequent behaviors that have close relations with Android system and regard them as sensitive behaviors. For the second challenge, we propose two semantic matching rules to bridge the gap between the sensitive behavior and the programming language based on the analysis of descriptions of Android concrete features (i.e., permissions, API calls and intents), as well as the keywords in the app code.

(i) We propose techniques that summarize the existing knowledge contained in magnanimity information of natural language documents and generate a novel type of features presented as verb-objective phrases that are easy to understand.

(ii) We propose two semantic matching rules that bridge the gap between the phrase-based features and programming language.

(iii) We design and implement CTDroid, an automatic feature engineering system. By using CTDroid, we construct a set of informative features that can be utilized for Android malware detection and familial classification.

(iv) We conduct extensive experiments to evaluate CTDroid on a large scale of real malware and benign apps. the experimental results show that CTDroid can achieve a 95.8% true positive rate with only 1% false positive rate for malware detection and a 97.9% accuracy for familial classification. Furthermore, our proposed features are more informative than those of state-of-the-art approaches.

The rest of this chapter is organized as follows. Section 6.2 details the construction of sensitive behavior. Section 6.3 reports the experimental results. Section 6.4 concludes the chapter.

6.2 Construction of Sensitive Behavior

6.2.1 Text Preprocessing

There are thousands of technical blogs on the web. It is neither effective (need an expert understanding of Android system) nor efficient (too much knowledge to learn) to carefully read each blog. To better automatically obtain significant knowledge from the blogs, we first transform the semantic meanings of blog contents into a set of behaviors. A behavior is represented as a tuple that consists of a verb and an object, and both of them are indispensable. The steps of extracting behaviors from blogs with NLP techniques are listed as below.

1) Sentence Extraction: Given a technical blog crawled from the website with HTML format, we initially use *jsoup* [23], a Java HTML parser, to extract the contents from the HTML file, and remove all non-ASCII symbols. Then we split the extracted content into a set of sentences with sentence segmentation.

2) Part-of-speech (POS) Tagging: For each extracted sentence, its typed dependency representations of the plain text in the form of *rule(gov, dep)* are extracted by using *Stanford typed dependency parser* [122, 27], a program that works out the grammatical structure of sentences. The *gov* and *dep* denote the governor word and the dependent word, respectively. The *rule* denotes the relation between the *gov* and *dep*. There are many different kinds of *rules* defined in the parser, such as *conj*, *det*, *dobj*, and *nsubjpass*. By carefully reading ten technical blogs, we find that most of the extracted subjects are the malware samples. Therefore, we do not consider the rules of which the *dep* is the subject. Moreover, given that we aim to extract the information that depicts the malicious activities conducted by malware, the rules such as *det* and *conj* that depict the definition and conjunction relationships cannot be used to find the malicious activities. Thus, we only focus on two main types of *rules*: *dobj* and *nsubjpass*. The *dobj* denotes that the *dep* is the (accusative) object of the *gov*. The *nsubjpass* denotes that the *dep* is the syntactic subject of the *gov* in a passive clause.

As listed in Table 6.1, after the decomposition of the plain text, we can get the corresponding typed dependency representations with the *gov* (i.e., “open”, “call” and

Table 6.1: An example of behavior extraction

plain text	“These instructions can be used to open a web page, call a phone number, or send an SMS text message to a premium number.”[1]

typed dependency representations	dobj(open, page)
	dobj(call, number)
	dobj(send, message)

behaviors	open—> web page call—> phone number send—> sms text message

“send”) and the *dep* (i.e., “page”, “number” and “message”). We construct one behavior for each generated typed dependency representation, where the *gov* is used as the *verb* and the *dep* is used as the *object*. Furthermore, we extend the *verb* and the *object* to their corresponding noun phrases by adding the adjective modifiers and identifying muti-word expressions. For example, the *object* “message” is extended to its noun phrase, i.e., “sms text message”.

3) Word Stemming: The noun phrases with similar semantic meaning would appear in different variants, such as “a phone number” and “phone numbers”. To address this problem, we first remove the stop words, the common words that would appear to be of little value for NLP analysis. The stop words used in our work is provided by [18], such as “a”, “an”, and “the”. Then we apply *WordNet* [30] to reduce the words based on their POS tag to their root forms. For example, the object “numbers” in its plural form would be reduced to “number”.

Then, given that different verbs would contain the similar meanings such as “get” and “return”, we regard these verbs as the same one. To this end, we manually construct 14 semantic groups based on a set of commonly used verbs provided by Anton *et. al* [33]. Then we add their similar verbs returned by *WordNet*. As listed in Table 6.2, each semantic group consists of a set of similar verbs and one representative verb. If the verb

Table 6.2: 14 semantic groups and their representative verbs.

Representative Verb	Similar Verbs
send	push, upload
get	return, obtain, collect, gather, gain, access
check	verify, confirm
connect	direct, redirect, open, call, contact
use	invoke, perform, run, execute, activate, conduct
cipher	encrypt, encode
decipher	decrypt, decode
delete	remove, wipe
prevent	abort, restrict, cease, disrupt, intercept
change	replace, modify, alter
set	reset
generate	create, build, make
store	write, remember, impress
download	load, install, deploy

of a behavior belongs to one of the semantic group, then it will be replaced with the corresponding representative verb. For example, the behavior “return—> phone number” will be changed to “get—> phone number”.

6.2.2 Extraction of Sensitive Behavior

After the text preprocessing of the collected blogs, 208K behaviors are extracted. However, we observe that most of the extracted behaviors present little significance for malware analysis. For example, the behavior “advise—> user” occurs when the researchers give some advice to the users about how to protect their smartphones. However, this behavior has little value for malware analysis. Thus, we propose a clustering-based approach to filter out the useless behaviors and mine the frequent behaviors that have close relations with Android system. These behaviors are regarded as the sensitive behaviors. To this end, we need to first propose an effective and efficient behavior similarity calculation method since there are too many extracted behaviors.

1) Behavior Similarity Calculation: For convenience, we use BH to denote the set of extracted behaviors. $BH = \{bh_i = (verb_i, object_i) | 1 \leq i \leq K\}$, where K is the total number of behaviors. Each behavior bh_i contains a $verb_i$ and an $object_i$. The similarity between two behaviors bh_i and bh_j depend on the similarities between their corresponding $verbs$ and $objects$, which are represented as $sim(verb_i, verb_j)$ and $sim(object_i, object_j)$, respectively. The similarity between bh_i and bh_j is obtained as Eq. 6.1.

$$sim(bh_i, bh_j) = \alpha * sim(verb_i, verb_j) + (1 - \alpha) * sim(object_i, object_j) \tag{6.1}$$

The parameter α is used to control the weights of the similarity of $verbs$ and $objects$; $0 \leq \alpha \leq 1$. The reason of introducing α is that if the behaviors whose *verbs* are general words such as "use", "get" and "return", their similarities would mainly rely on the $sim(object_i, object_j)$ rather than $sim(verb_i, verb_j)$. To this end, we assign different weights to the $verbs$ to denote their importance in the similarity calculation. Specifically, if a $verb$ is generally used in our extracted behaviors, then its weight should be low. Thus, we use the inverse document frequency (IDF) to measure the inverse frequency of $verb$ that appears across all the behaviors. Therefore, the weight of $verb_i$ is calculated as:

$$w(verb_i) = \log_2 \frac{K}{Num(bh_{verb_i})}, \tag{6.2}$$

where $Num(bh_{verb_i})$ denotes the number of behaviors that contain $verb_i$. Then, all the $w(verb)$s are normalized between 0 and 1. Finally, for the $sim(bh_i, bh_j)$, its α is obtained as:

$$\alpha = \frac{w(verb_i) + w(verb_j)}{2} \tag{6.3}$$

Next, to calculate the similarity between $verbs$ or $objects$ which are actually phrases (phs), we first transform them into a calculable form. Here we rely on the tool called *Word2Vec* [97]. *Word2vec* takes a large corpus of text as its input and produces a vector space, with each unique word in the corpus being assigned a corresponding vector in the space. In our work, we collect a 12.2G corpus from Wikipedia [12] and put them into *Word2vec* with the skip-gram model [73]. Each word wd in the corpus is represented as a vector with l dimensions as Eq. (6.4); $l = 100$ in our work.

$$\overrightarrow{vec(wd)} = \langle v_1, v_2, \ldots, v_l \rangle \tag{6.4}$$

As introduced in [97], semantic relations among words can be captured via simple vector operation. For example, $\overrightarrow{vec(\text{“}better\text{”})} - \overrightarrow{vec(\text{“}good\text{”})} \approx \overrightarrow{vec(\text{“}faster\text{”})} - \overrightarrow{vec(\text{“}fast\text{”})}$, in which the minus sign denotes vector substraction operation. Leveraging the characteristic of vector operation in *Word2Vec*, we obtain the the vector of a phrase ph by the vector adding operation on all the words in ph as Eq. (6.5). The cosine similarity is widely used to find the similarity between two given vectors. Thus the similarity between two phrases can be calculated with cosine similarity based on Eq. (6.6), in which $||\overrightarrow{vec}||$ is the Euclidean norm of the vector $\overrightarrow{vec}$.

$$\overrightarrow{vec(ph)} = \sum_{wd \in ph} \overrightarrow{vec(wd)} \tag{6.5}$$

$$cosine(\overrightarrow{vec(ph_1)}, \overrightarrow{vec(ph_2)}) = \frac{(\overrightarrow{vec(ph_1)} \times \overrightarrow{vec(ph_2)})}{||\overrightarrow{vec(ph_1)}|| \cdot ||\overrightarrow{vec(ph_2)}||} \tag{6.6}$$

3) Behavior Clustering: Based on the similarity calculation of behaviors, we mine the frequent behaviors via the clustering of behaviors. Algorithm 5 lists the step of behavior clustering with the input of all generated behaviors $BH = \{bh\}$ and two threshold values, θ and ϵ. θ denotes the similarity threshold between a behavior and a cluster. In other words, if the average similarity between a behavior bh_i with all the behaviors in cluster c_j, represented as $\overline{sim}(bh_i, c_j)$, is higher than θ, then the behavior bh_i is added into the cluster, indicating that the behavior bh_i has very close semantic meanings with those in

Algorithm 5 Clustering of behaviors

```
Require:
    $BH = \{bh\}$   // $BH$ denotes the set of extracted behaviors in blogs.
    $\theta$   // $\theta$ denotes the similarity threshold value of adding behaviors into clusters.
    $\epsilon$   // $\epsilon$ denotes the support threshold value of filtering out clusters.
Ensure:
    $C$   // $C$ denotes the set of output clusters and each cluster contains a set of similar
    behaviors.
 1: $p = 1, c_1 = \{bh_1\}, C = \{c_1\}$
 2: for each $bh_{i,i\neq 1}$ in $BH$ do
 3:     $c' = argmax_{c_j \in C}\, \overline{sim}(bh_i, c_j)$
 4:     if $\overline{sim}(behav_i, c') \geq \theta$ then
 5:         $c' = c' \cup \{behav_i\}$
 6:     else
 7:         $p = p + 1, c_p = \{behav_i\}, C = C \cup \{c_p\}$
 8:     end if
 9: end for
10: for each $c_j$ in $C$ do
11:     if $sup(c_j) < \epsilon$ then
12:         $C.remove(c_j)$
13:     end if
14: end for
15: return $C$
```

cluster c_j. ϵ denotes the support threshold value of grouped clusters. If the support value of cluster c_j, represented as $sup(c_j)$ is less than ϵ, we filter out this cluster.

In Algorithm 5, C is initialized with only one cluster $c_1 = \{bh_1\}$ (line 1). Then all the other behaviors in BH are successively calculated to check whether there exists a cluster in C that the current behavior can be added in (lines 2-9). After that, we filter out the clusters whose support values are less than ϵ (line 10-14).

After the clustering of behaviors, we can obtain a set of clusters C and each cluster $c \in C$ contains a set of similar behaviors. In each cluster, the behavior with the highest frequency number is selected as the representative behavior $repBh$. The frequency number of a behavior denotes the times of the behavior occurs in BH. Table 6.3 lists an example of the generated cluster, in which all behaviors contain the similar semantic meanings of sending text messages. The number attached to the behavior denotes its corresponding frequency number.

Table 6.3: An example of behavior cluster.

Representative behavior	send—> text message
Behaviors	send—> text message (236) send—> premium text message (3) send—> multiple text message (2) send—> sms text message (2) send—> one text message (2) ......

Table 6.4: Android system related concepts [60].

phone number	photo	imei	password	camera
phone call	time	contact	radio	email
device id	keylock	pin	bookmark	calendar
serial number	network	account	file	package
subscriber id	location	browser	shortcut	screenshot
text message	battery	alarm	wallpaper	bluetooth
microphone	command	permission	activity	wifi

To identify the sensitive behaviors, we filter out the behaviors with little significance for malware analysis within two steps.

- First, we remain the behaviors whose verbs belong to our constructed 14 semantic groups, since most other verbs are too general to identify their concrete actions in app code such as “protect”, “alert”, and “infect”.

- Second, we remain the behaviors that have close relations with Android system. To this end, we obtain a set of Android system sensitive concepts based on the work of Felt *et. al* [60], in which they conduct a research for the user concerns about 99 smartphone risks. As a result, there are 35 sensitive concepts listed in TABLE 6.4.

6.2.3 Feature Construction

After the generation of sensitive behaviors, it is non-trivial to directly utilize the sensitive behaviors for malware analysis with machine learning algorithms due to the semantic gap between the sensitive behaviors and the programming language. To address this challenge, we propose two semantic matching rules by leveraging the descriptions of Android concrete features (i.e., permissions, API calls and intents), as well as the keywords in the app code.

APK Disassembling: Introduced as before, with existing mature disassembling tools such as *apktool* [16], we are able to obtain the `AndroidManifest.xml` file and the dalvik code files. The `AndroidManifest.xml` file contains essential information about an app to the Android system, including the requested permissions and intents. It is worth noting that the widely used third-party and advertisement libraries might affect the performance of malware analysis. We filter out these libraries from the dalvik code by using the list provided by [88, 87].

Feature Vector Construction: Basically, the dalvik code is the main part of an app that we need to match with our sensitive behaviors. Furthermore, existing approaches [137, 61, 157] reveal that permissions and intents are significant for malware analysis. Thus, we also match such concrete features (i.e., permissions and intents) with our sensitive behaviors. Our two matching rules are introduced as below.

Rule I: Matching with permissions and intents.

In our work, 140 permissions and 261 intents are collected from the Android document [13]. However, it is not effective to directly match the permissions and intents with the sensitive behaviors because of the insufficient literal meanings. Therefore, the corresponding descriptions of the permissions and intents are also collected to provide

Table 6.5: An example of permission matching

Sensitive behavior	send—>text message
Permission	SEND_SMS
Description	It allows an application to send SMS messages.
Extracted behavior	send—>sms message (sim: 0.98)

Table 6.6: An example of intent matching

Sensitive behavior	check—>package
Intent	PACKAGE_VERIFIED
Description	Send to the system package verifier when a package is verified.
Extracted behavior	verify—>package (sim: 1.0)

useful information. To match the concrete permissions and intents with given sensitive behavior, the collected descriptions are parsed into behaviors. In addition, if the name of a permission or an intent consists of a verb and an object, one more behavior is constructed. Then the extracted behaviors are matched with the given sensitive behavior by using our similarity calculation method. If there exists a similarity that is higher than the preset θ, then we define that the app contains the current sensitive behavior feature.

TABLE 6.5 and TABLE 6.6 list examples of permission matching and intent matching, respectively. The number behinds the extracted behavior denotes its similarity with the sensitive behavior. It is worth noting that in TABLE 6.6, since the verb "verify" and the verb "check" belong to the same semantic group listed in TABLE 6.2, "verify" is replaced with "check" and the similarity between the two behaviors is 1.0.

Rule II: Matching with dalvik code.

Dalvik code is a human readable representation of the binary bytecode. From the generated dalvik code files we initially extract the method bodies by recognizing the identifies *.method* and *.end method*. In each method body, API calls are invoked to perform

Table 6.7: An example of API call matching.

Sensitive behavior	get—>phone number
API	getLine1Number()
Description	It returns the phone number for line 1, for example, the MSISDN for a GSM phone.
Extracted behavior	return—>phone number (sim: 1.0)

specific behaviors. For example, the API call `getLine1Number()` is used to return the phone number of the device, which can be matched with the "get—> phone number" behavior. However, expert knowledge is needed to link the meanings of line1 number with the phone number. Similar to permissions and intents, we also leverage the descriptions of API calls to help us match them with given sensitive behaviors. TABLE 6.7 lists an example of API call matching.

Unfortunately, not all the identified sensitive behaviors have corresponding successfully matched API calls. For example, Fig. 6.1 presents the snippets of deleting a text message, which is implemented by using the *ContentResolver:delete()* function with the argument of parsed string *content://sms/conversations/*. Directly matching the given sensitive behavior with the invoked API call in the method body is not sound for such cases.

To address this problem, the method body is initially tokenized into a bag of words. For example, *ContentResolver:delete()* is tokenized as {"content", "resolver", "delete"}. Then we check whether each word in the sensitive behavior is contained in the word bag. In this way, the snippets in Fig. 6.1 are matched with "delete—> text message" sensitive behavior based on the matched words "delete" and "sms" (similar meaning as "text message") that are marked in red.

Next, to perform malware analysis with machine learning algorithms, each app should be represented as a feature vector. Specifically, for a set of n given apps $X = \{x_1, x_2, \ldots, x_n\}$ and a set of k identified sensitive behavior feature $F = \{f_1, f_2, \ldots, f_k\}$,

```
const-string v3, "content://sms/conversations/"
......
invoke-static {v2}, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
move-result-object v2
const/4 v3, 0x0
const/4 v4, 0x0
invoke-virtual {v0, v2, v3, v4}, Landroid/content/ContentResolver;-
>delete(Landroid/net/Uri;Ljava/lang/String;[Ljava/lang/String;)I
......
```

Figure 6.1: Snippets of deleting a text message.

each app x_i is represented as a feature vector $\mathbf{x_i} = \langle x_{i1}, x_{i2}, \ldots, x_{ik} \rangle$, where x_{ij} denotes the value of the j^{th} feature for the i^{th} app. x_{ij} is calculated based on the above two matching rules in algorithm 6.

In algorithm 6, x_{ij} is initially set as 0 (line 1). Then we extract a required permission set Per_{x_i} and an intent set Int_{x_i} from the *AndroidManifest.xml* file of app x_i. After that, we match each permission and intent in the two sets with the given sensitive behavior f_j with matching rule I, and increase the feature value with 1 if there exists a successful matching (lines 2-9). Next, we construct a method set $Method_{x_i}$ by extracting the methods from the dalvik code, and match each method with f_j with matching rule II (lines 10-21). Note that our features are different from the binary features (e.g., permissions and API calls) that are set as 1 or 0, we not only consider the occurrence of corresponding sensitive behavior but also calculate its frequency of occurrence. By doing so, the feature vector constructed for each app contains more information than those constructed based on binary features.

Finally, we conduct two malware analysis tasks, malware detection and familial classification. Note that the labels attached to the feature vectors for the two tasks are different. For the task of malware detection, there are two types of labels, malicious and benign, which are denoted as 1 and 0 respectively. In other words, if a given app x_i is a malicious one, then its corresponding label y_i is set as 1, or the label is set as 0 if the app is benign. However, for the task of malware classification, the label y_i belongs to one of the

Algorithm 6 Calculation of feature value

```
Require:
    $x_i, f_j$    // $x_i$ denotes the $i^{th}$ app and $f_j$ denotes the $j^{th}$ sensitive behavior feature.
Ensure:
    $x_{ij}$    // $x_{ij}$ denotes the output feature value.
 1: $x_{ij} = 0$
 2: $Per_{x_i} = \{per\}$    // $Per_{x_i}$ denotes the required permission set of $x_i$.
 3: if $\exists per \in Per_{x_i}$ and $MatchingRule - I(per, f_j)$ then
 4:     $x_{ij} + +$
 5: end if
 6: $Int_{x_i} = \{int\}$    // $Int_{x_i}$ denotes the used intent set of $x_i$.
 7: if $\exists int \in Int_{x_i}$ and $MatchingRule - I(int, f_j)$ then
 8:     $x_{ij} + +$
 9: end if
10: $Method_{x_i} = \{md\}$    // $Method_{x_i}$ denotes the method set of $x_i$.
11: for each $md$ in $Method_{x_i}$ do
12:     $API_{md} = \{api\}$    // $API_{md}$ denotes the API call set of $md$.
13:     if $\exists api \in API_{md}$ and $MatchingRule - II(api, f_j)$ then
14:         $x_{ij} + +$
15:     else
16:         $WdBag_{md} = \{wd\}$    // $WdBag_{md}$ denotes the word bag of $md$.
17:         if $MatchingRule - II(WdBag_{md}, f_j)$ then
18:             $x_{ij} + +$
19:         end if
20:     end if
21: end for
22: return $x_{ij}$
```

malware family names, such as *geinimi* or *droidkungfu*. Therefore, for each task a dataset is initially constructed and represented as $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$. Then, the dataset is split into a training dataset and a testing dataset. By applying known machine learning algorithms on the training dataset, different classifiers are generated. After that, each sample x_i in the testing dataset will be fed into the classifier and a label y_i' will be returned. If y_i' is equal to y_i, then the sample is correctly classified with the generated classifier, or it is wrong.

6.3 Evaluation of CTDroid

To evaluate CTDroid, we first introduce the study setup of our experiments, and then address the following five research questions.

RQ 1 *Which classifier and parameters (i.e., θ and ϵ) are appropriate for CTDroid? (Section 6.3.2)*

RQ 2 *Can CTDroid detect Android malware with high true positive rate and low false positive rate? (Section 6.3.3)*

RQ 3 *Can CTDroid classify Android malware into their correct families with high accuracy? (Section 6.3.4)*

RQ 4 *Can CTDroid handle a large scale of apps with high efficiency? (Section 6.3.5)*

RQ 5 *To what extent is CTDroid resistant to code obfuscation techniques? (Section 6.3.6)*

6.3.1 Study Setup

CTDroid analyzes two main types of datasets: 1) Technical blogs, which contain the natural language contents about Android malware. 2) Android malware and benign apps, which are used to evaluate the performance of CTDroid for malware detection and familial identification.

Technical Blogs: In general, the technical blogs are written by researchers with specialized knowledge. Therefore, we utilize the contents in the technical blogs to mine sensitive behaviors that might do harmful activities to users potentially. The corpus of technical blogs is crawled from ten websites, including nine security companies websites [6, 9, 10, 11, 24, 7, 5, 8, 4] and the well known personal website of Jiang [2], from 2010 to 2017. Given that we focus on Android malware analysis, we use the keywords such as “Android”, “malware”, and “malicious” to filter out the irrelevant blogs. We pick these ten security websites because of their expert analysis on Android malware and we believe in their analysis result described in the crawled technical blogs. In summary, we

Table 6.8: Descriptions of collected technical blogs.

Web-site	# Blogs	Web-site	# Blogs
Fortinet [6]	103	Lookout [7]	145
Secure List [9]	179	Cheetah Mobile [5]	50
Security Intelligence [10]	142	Palo Alto [8]	39
Trend Micro [11]	220	Check Point [4]	155
McAfee [24]	324	Jiang [2]	28

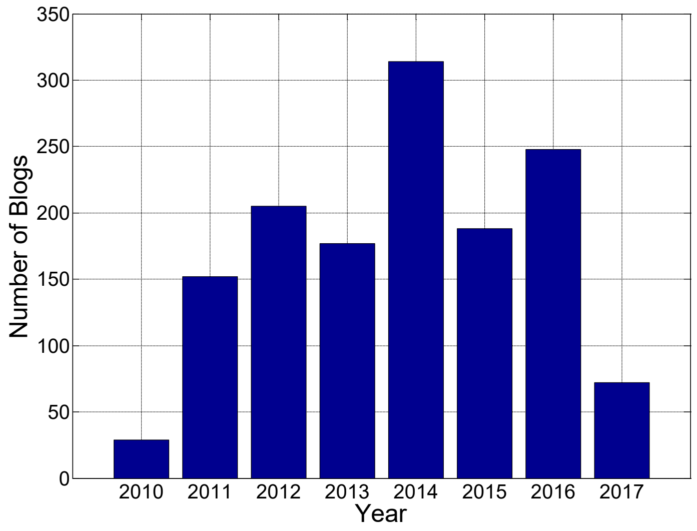


Figure 6.2: Time distribution of collected technical blogs.

collect 1,385 Android malware related technical blogs that are listed in TABLE 6.8. The time distribution of the collected blogs is illustrated in Fig. 6.2.

Android Malware and Benign Apps: To evaluate the performance of CTDroid for malware detection and familial identification, we apply it on four malware datasets, including three widely-used datasets provided by Gnome project [156], Drebin [34], and FalDroid, and a new dataset constructed by ourselves by collecting recent malware samples from Palo Atlo [8]. Specifically, for malware detection, we collect an equal number of most popular (10,000+ downloads) benign apps from Google Play [20] in the same period and add them to the four provided malware datasets. Each benign app has

Table 6.9: Descriptions of three datasets used for malware detection (MD).

Dataset	#Malware	#Benign Apps
MD-I	1,260	1,260
MD-II	5,560	5,560
MD-III	8,407	8,407
MD-IV	1,015	1,015

Table 6.10: Descriptions of three datasets used for familial identification (FI).

Dataset	#Malware	#Families
FI-I	1,247	33
FI-II	5,513	132
FI-III	8,407	36
FI-IV	1,015	69

been uploaded to the VirusTotal to make sure that no virus engine reports it as malicious. Therefore, four datasets that contain both malware and benign apps are constructed for malware detection (MD). For convenience, the four datasets are named as MD-I, MD-II, MD-III, and MD-IV, and their descriptions are listed in TABLE 6.9. For familial identification (FI), given that we need to split each dataset into a training set and a testing set, we remove the malware families that contain only one sample. For convenience, the four datasets used for familial identification are named as FI-I, FI-II, FI-III, and FI-IV, and their descriptions are listed in TABLE 6.10.

Metrics: For malware detection, TPR is used to denote the percentage of malware that are correctly predicted as malware, and FPR is used to denote the percentage of benign apps that are incorrectly predicted as malware. The goal of any malware detection research is to achieve a high value for TPR and a low value for FPR. For familial identification, the term classification accuracy is used to denote the percentage of malware that are correctly classified into their corresponding families.

Baseline Approaches: We compare the performance of CTDroid in malware detection and familial identification with three baseline approaches, i.e., FeatureSmith [157], FalDroid, and MaMaDroid [95]. The descriptions of the three baseline approaches are listed as below:

- Zhu and Dumitras proposed FeatureSmith [157], which first identifies 173 concrete features, including permissions, API calls, and intents, which occur in scientific papers. Then they extract the identified features from the `AndroidManifest.xml` file and dalvik code for malware detection.
- We proposed FalDroid [59] in Chapter 4, which first constructs fregraphs from the malware samples within the same family to denote their common malicious behaviors. Then they regard each fregraph as a feature and construct a feature space for malware analysis.
- Mariconti *et al.* proposed MaMaDroid [95], which first builds a behavioral model in the forms of a Markov chain from the sequence of extracted API calls performed by apps. Then it extracts features from the Markov chain to perform malware analysis.

6.3.2 Selection of Classifier and Arguments

To choose the appropriate classifier for CTDroid, five different machine learning algorithms, including decision tree [120], k-nearest neighbours (k-NN) [32], logistic [83], multi-layer perceptron (MLP) [119] and random forest [43], are applied in our approach. Specifically, we construct five corresponding classifiers based on these algorithms and apply them for malware detection on the MD-III dataset. Note that here we initially set our two important parameters, i.e., θ and ϵ, as 0.9 and 3, respectively.

Fig. 6.3 illustrates the malware detection performance of CTDroid on MD-III dataset

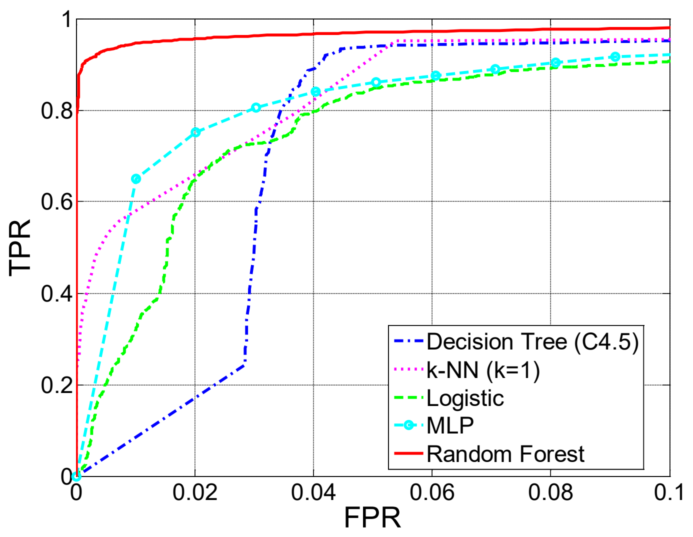


Figure 6.3: Detection performance of CTDroid on the MD-III dataset with five different classifiers.

with five different classifiers. The result shows that random forest outperforms the other four classifiers. When FPR is 0.01, the TPR of random forest can achieve 0.947, much higher than those of the other classifiers. Therefore, due to the superior performance of random forest among the five classifiers, random forest is selected as our default classifier in later experiments.

Next, we investigate the influence of θ and ϵ to our performance. θ controls the similarity calculation between extracted behaviors; ϵ controls the threshold value of filtering out useless clusters. To set an appropriate θ, we manually construct a set of behaviors with similar meanings and then calculate their similarities between any two behaviors. We find that all the calculated similarities are higher than 0.9. Thus, θ is set as 0.9 in our work. ϵ is a parameter to balance the size of feature space and detection performance. The higher of ϵ, the fewer sensitive behavior features will be, but we might miss some significant features if ϵ is too high. However, if the ϵ is too low, we might introduce some useless features. To select an appropriate ϵ, we vary the values of ϵ as $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 100\}$. The TPR values (FPR=0.01) of CTDroid and numbers of

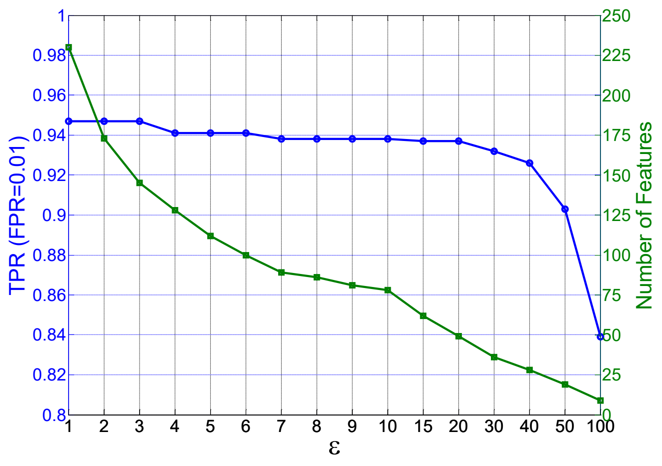


Figure 6.4: TPR values (FPR=0.01) of CTDroid on the MD-III dataset and numbers of generated sensitive behavior features with different ϵ.

generated sensitive behavior features with different ϵ are illustrated in Fig. 6.4. We find that with the increase of ϵ, the number of features decreases. Moreover, the TPR value starts to decrease when ϵ is higher than 3. Therefore, to achieve high performance, ϵ is set as 3.

6.3.3 Accuracy of Malware Detection

To answer RQ 2, we evaluate the malware detection performance of CTDroid on four datasets and compare it with three baseline approaches, i.e., FeatureSmith [157], FalDroid, and MaMaDroid [95]. Specifically, for each dataset, we train four random forest classifiers but four different feature sets. In our work, when $\epsilon = 3$, 145 features are extracted from collected blogs. For FeatureSmith, 173 features are identified from scientific papers. However, for FalDroid, its feature space is constructed from the training dataset, thus the feature number of FalDroid varies from different datasets. For MaMaDroid, 121 features are extracted.

Fig. 6.5 presents the malware detection performance of CTDroid and three baseline

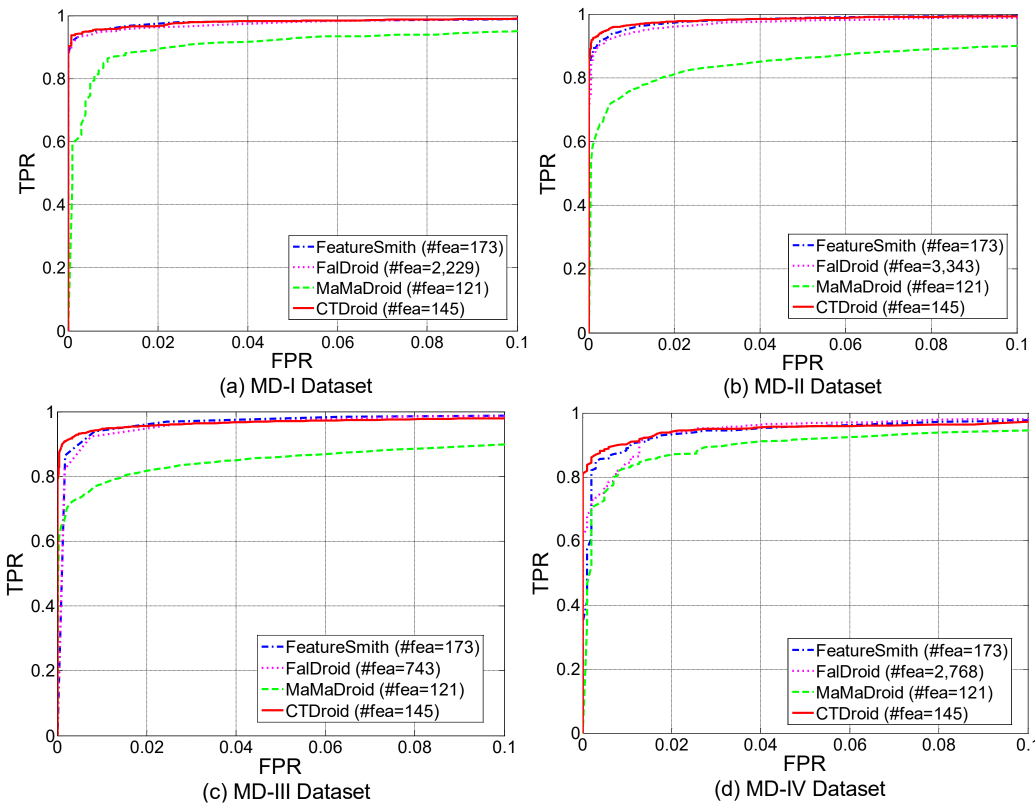


Figure 6.5: Malware detection performance of CTDroid and three baseline approaches with all features on four malware detection datasets.

approaches with their all features using receiver operating characteristic (ROC) plots. The plots illustrate the relationship between the TPRs and the FPRs of the four classifiers. Note that the term #fea in Fig. 6.5 denotes the number of extracted features. The results demonstrate that CTDroid gets an almost equally performance with FeatureSmith and FalDroid, and performs better than MaMaDroid on the four datasets. For example, on the MD-I dataset, when the FPR is 0.01, all of CTDroid, FeatureSmith, and FalDroid have a TPR around 0.958, while the TPR of MaMaDroid is only 0.869.

Even CTDroid gets a fairly good performance for malware detection when FPR is 0.01, it still incorrectly classifies about 13 benign apps as malware on the MD-I dataset. The main reason to explain the incorrectly classified samples is that some words contain multi-meanings, thus affecting the performance of semantic matching approach. For example, if

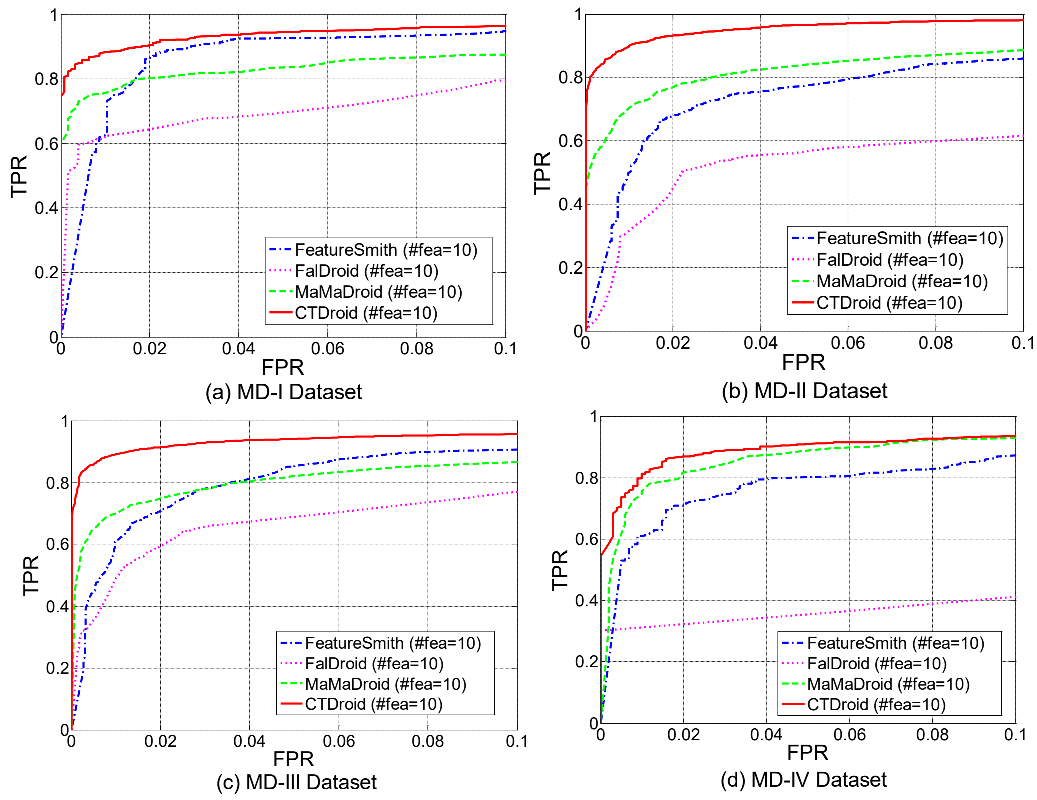


Figure 6.6: Malware detection performance of CTDroid and three baseline approaches with top-10 features ranked by information gain on four malware detection datasets.

a method body contains the string value "Please contact me", CTDroid would extract the wrong sensitive concept "contact" and incorrectly identify the related features from the method.

Recall that our proposed features not only consider the occurrence of corresponding sensitive behavior, but also calculate its frequency of occurrence, thus our features would contain more information than the binary features generated by FeatureSmith and FalDroid. To evaluate the effectiveness with few features, we compare the detection performance of CTDroid and baseline approaches with top-m features ranked by information gain [110].

Fig. 6.6 presents the detection performance of the four approaches with top-10 ($m =$

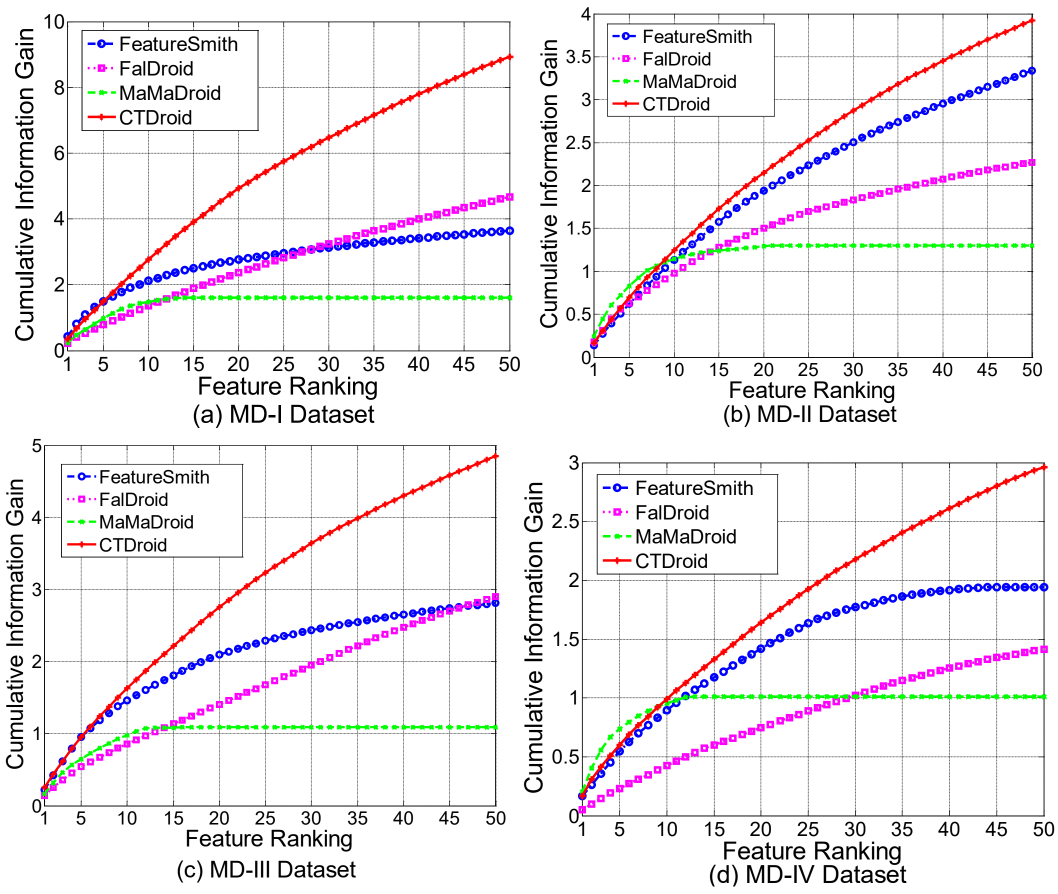


Figure 6.7: Cumulative information gain of features ranked by information gain on four malware detection datasets.

10) features ranked by information gain. The results on all the four datasets demonstrate that with only 10 features CTDroid outperforms the baseline approaches due to the more informative features. For example, on the MD-II dataset, when the FPR is 0.01, CTDroid gets a TPR of 0.9, while the TPRs of FeatureSmith, FalDroid and MaMadroid are 0.503, 0.340 and 0.710, respectively.

To further investigate the information gain of different features, we vary the values of m from 1 to 50 and calculate the cumulative information gain of the top-m ranked features. As illustrated in Fig. 6.7, on all the four datasets, the cumulative information gains of CTDroid are higher than those of baseline approaches. When m is set as 10, the

Table 6.11: Classification performance of CTDroid and three baseline approaches on four different datasets.

Dataset	Approach	#Fea	Accuracy	#Fea	Accuracy
FI-I	FeatureSmith	173	0.940	10	0.868
	FalDroid	2,229	0.972	10	0.740
	MaMaDroid	121	0.860	10	0.859
	CTDroid	145	**0.979**	10	**0.935**
FI-II	FeatureSmith	173	0.950	10	0.674
	FalDroid	3,343	0.953	10	0.623
	MaMaDroid	121	0.878	10	0.874
	CTDroid	145	**0.961**	10	**0.911**
FI-III	FeatureSmith	173	0.918	10	0.662
	FalDroid	743	**0.942**	10	0.592
	MaMaDroid	121	0.849	10	0.838
	CTDroid	145	0.922	10	**0.859**
FI-IV	FeatureSmith	173	0.883	10	0.606
	FalDroid	2,768	0.876	10	0.581
	MaMaDroid	121	0.839	10	0.833
	CTDroid	145	**0.890**	10	**0.848**

cumulative information gain of CTDroid is 1.243 on the MD-II dataset, more than those of FeatureSmith, FalDroid, MaMaDroid, i.e., 1.130, 0.978, 1.144. Moreover, we find that the cumulative information gain of MaMaDroid hardly changes when m is higher than about 15, indicating that most of the features extracted by MaMaDroid have little significance for malware analysis.

6.3.4 Accuracy of Familial Identification

To evaluate the familial identification performance of CTDroid, we apply it on the four datasets, i.e., FI-I, FI-II, FI-III, and FI-IV. Furthermore, we compare CTDroid with the three baseline approaches. The results are listed in Table 6.11, where the values marked in bold denote the highest classification accuracy for each dataset.

In Table 6.11, columns 3-4 list the classification performance of the three approaches

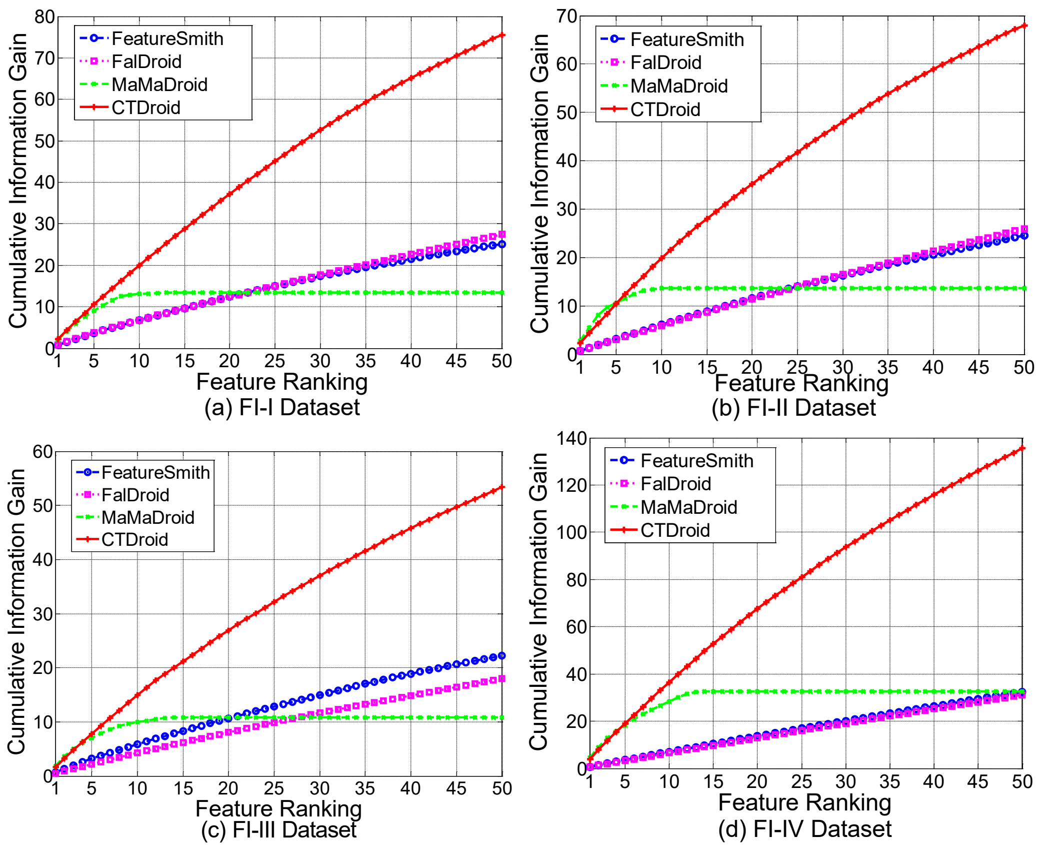


Figure 6.8: Cumulative information gain of features ranked by information gain on four familial identification datasets.

with their all features. With all the 145 sensitive behavior features, CTDroid performs best on FI-I dataset, FI-II dataset, and FI-IV dataset. However, on FI-III dataset, the accuracy of CTDroid is about 0.02 less than that of FalDroid. Columns 5-6 list the classification performance of the three approaches with top-10 features ranked by information gain. With only 10 features, CTDroid outperforms the baseline approaches. For example, on FI-III dataset, CTDroid can get an accuracy of 0.859, much higher than those of baseline approaches.

We further calculate the cumulative information gain of the three approaches when m varies from 1 to 50. As illustrated in Fig. 6.8, the results suggest that the cumulative information gains of CTDroid on all the four datasets are much higher than those of baseline approaches, indicating that our proposed features are more informative compared with those of baseline approaches.

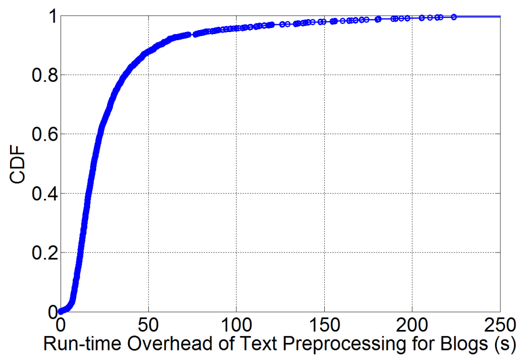
6.3.5 Analysis of Run-time Overhead

To answer RQ 4, we investigate the run-time overhead of CTDroid. In this work, text preprocessing for blogs and feature vector construction for apps are the two main procedures that require more computation resource than the other two procedures. The cost of text preprocessing and feature construction depends on the number of collected blogs and the number of apps, respectively.

The CDF of run-time overhead for the two procedures are illustrated in Fig. 6.9. The left figure presents that 91.3% blogs require less than 60s to extract the behaviors from their content by using *Stanford Parser*. In total, 11h is required to implement the text preprocessing for all the 1,385 collected blogs. The right figure presents that 0.5s is needed on average to construct a feature vector for each given app after the disassembling. The cost of disassembling of apks is the same as the other approaches since it is the necessary step to statically analyze apps for all the three approaches. It is worth noting that the text preprocessing and the feature vector construction procedures could be conducted on several PCs in parallel, thus further reducing the total run-time overhead.

For the sensitive behavior generation procedure, with a set of about 208K extracted behaviors as input, 10min is needed for the clustering of behaviors and outputting the set of sensitive behaviors. Finally, the construction of random forest classifier used for malware analysis requires less than 1min.

We also investigate the efficiencies of the three baseline approaches. For FeatureSmith, its run-time overhead is similar to ours, since both of the two approaches process feature



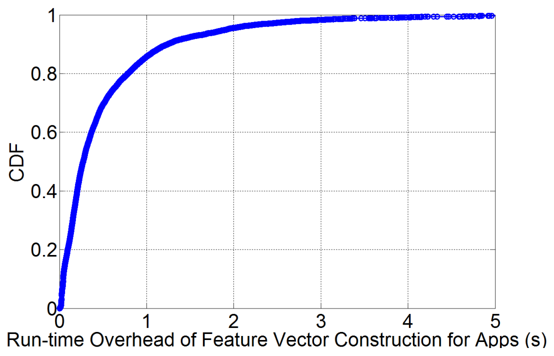


Figure 6.9: CDFs of the run-time overhead for text preprocessing and feature vector construction.

engineering from contents written in natural language. However, FalDroid relies on graph analysis that requires about one week to construct the fregraph-based feature space. Furthermore, more than 2s is needed to generate the feature vector for a given app after the disassembling. For MaMaDroid, about two days are needed to construct all the call graphs and extract feature vectors for all the apps.

6.3.6 Analysis of Resilience

To evaluate the resilience of CTDroid to code obfuscation techniques, we only consider the techniques that try to increase the values of sensitive behavior features, since the technique of deleting code that reduces the feature values might affect the functionalities of original apps. For example, the code obfuscation techniques can add the value of feature "send—>text message" from 0 to 1, but hard to reduce it from 1 to 0 without affecting the app's functionality of sending messages.

Specifically, the inserting of useless instructions might increase the feature values. For example, if the attacker inserts a string "We will send a text message" into a method, our approach will incorrectly match the "send—>text message" feature. This technique might misguide CTDroid to classify a benign app as malicious, but can hardly misguide

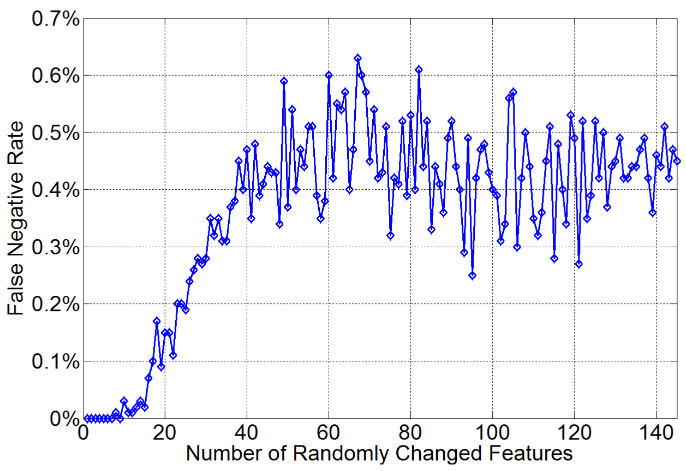


Figure 6.10: FNRs of CTDroid for the obfuscation of increasing feature values by 1.

a malware into benign. To evaluate the resilience of CTDroid to such technique, we first randomly select 100 malware from the MD-I dataset as testing set. Then we increase the values of t randomly selected features with 1. After that, we fed these obfuscated feature vectors into constructed classifier to detect whether their corresponding output labels are still 1, indicating that the obfuscation techniques do not affect the detection result. We vary the t from 1 to 145 and repeat this experiment 100 times. The results are shown in Fig. 6.10, where the false negative rate (FNR) denotes the percent of malware samples that are incorrectly classified as benign after the changing of feature vectors. We observe that when t is less than 20, nearly no malware is incorrectly classified. The highest FNR is 0.63%, indicating that on average less than 1 malware sample in the testing set is affected by the obfuscation techniques.

Moreover, we also vary the increased feature values as $\{1,2,3,4,5\}$. Note that here we fix the number of t as 20. Table 6.12 lists the average FNRs with different increased feature values from 1 to 5. The results demonstrate that with the increase of feature values, the FNR increases. The main reason is that if the feature value changes a lot, the corresponding sample will be regarded as an abnormal one, thus causing the increase of FNR. To limit the affect caused by inserting useless instructions, it is a promising way to combine dynamic

Table 6.12: FNRs of CTDroid for the obfuscation of increasing feature values from 1 to 5.

Increased Feature Value	FNR
1	0.12%
2	0.28%
3	0.35%
4	0.38%
5	0.6%

analysis techniques with our approach to filter out the code that would never be executed.

6.4 Brief Summary

In this chapter, we propose a novel system, CTDroid, to automatically construct informative sensitive behavior features for malware analysis by analyzing a corpus of Android malware related technical blogs. To evaluate the effectiveness of constructed features, we evaluate CTDroid for two tasks, i.e., malware detection and familial identficaition. Our extensive evaluation results show that CTDroid can achieve high accuracy and efficiency. Furthermore, our features present more information than those of binary features proposed by the state-of-the-art approaches.

Chapter 7

Conclusion

7.1 Conclusion

Android has become the most popular mobile OS. However, it has also become the major target of Android malware. The rapid increase in the number of Android malware poses great threats to the smart phone users, such as financial charge, information collection, and remote control. Thus, the in-depth study of the security issues of mobile applications is of great significance to the sound development of the smart phone ecosystem. To further study two sub problems in mobile security, i.e., malware detection and familial identification, two kinds of behavior models and four different types of features are proposed from the novel perspective of feature engineering.

First, to overcome the low accuracy and efficiency problems caused by the morphological diversity of malicious code, the sensitive subgraph is first constructed as our analysis model. Based on the sensitive subgraph, for malware detection, a structure-based feature called maximum sensitive subgraph is proposed to depict the most sensitive behavior of a given app. Based on the proposed feature, this thesis designs and implements DAPASA, an approach that detects Android piggybacked apps. DAPASA can not only detect the piggybacked apps dependently but also has the ability to complement permission- and API-based approaches from a new perspective of the invocation structure.

Second, for family identification, a new feature called fregraphs is proposed to represent the common behaviors of malware samples that belong to the same family. Then, this thesis designs and implements FalDroid, an approach that automatically classifies Android malware into their corresponding families and selects representative malware samples in each family accordance with fregraphs. In this way, FalDroid can effectively reduce the analytical workload and accelerate malware analysis.

Third, to overcome the limitation of existing supervised learning approaches in handling unlabeled dataset, the graph structure of sensitive subgraph is abstracted by leveraging the graph embedding techniques and a new feature called SRA is proposed to depict the similarity relationships of structural roles of sensitive API call nodes in a graph. Then this thesis designs and implements GefDroid, an approach that constructs a malware link network to depict the similarity relationships between all samples based on the SRA feature. In this way, this study can handle the unlabeled samples with unsupervised learning.

Finally, to ease the labor-intensive manual feature engineering process, this study proposes techniques that summarize the existing knowledge contained in magnanimity information of natural language documents and generates a novel type of features called sensitive behavior. This thesis designs and implements CTDroid, an automatic feature engineering system. By using CTDroid, a set of informative features is constructed from technical blogs that can be utilized for Android malware analysis.

7.2 Future Work

As we have done four major works related to malware detection and familial identification in this thesis, future work also lies in the following directions.

1) Native Code: In this thesis, we limit our analysis to the FCG model constructed based on the dalvik code. We do not analyze native code. Thus, our approach would miss the malicious behaviors implemented in native code. However, there are many binary analysis frameworks, such as Angr [124], that can help us address this limitation by constructing the FCG of the native code. Then, we could apply our approach to conducting similarity detection of such FCGs. We will explore this approach in future work.

2) Multi-label Malware: GefDroid can well handle the samples in our used datasets from which each sample belongs to exactly one malware family. However, it might fail when dealing with the multi-label malware samples that contain code from multiple malware families. The multi-label malware samples belong to the overlapping region in the constructed MLN, which might be handled by the overlapping community detection algorithms. We leave the detection of multi-label samples as our future work.

3) Third-party Libraries: To remove the third-party and advisement libraries, we extend the widely-used library list by adding the class names of 5,000 benign apps. Even the list works well on our datasets, it is unclear how does the list performs when applying our approaches on other datasets. In future work, we plan to construct more big datasets that contain recent malware samples and evaluate the performance of our approaches on them.

4) Sensitive API Calls: Our detection of sensitive API calls relies on the set provided by SuSi[111], which now, four years later, might be incomplete or outdated. Missing or incorrect sensitive API calls contained in SS would make our approaches miss or misidentify the common malicious behaviors between malware samples within the same families. Furthermore, since the sensitive API calls are extracted statically in our approaches, the ones that are never executed by the malware samples would introduce noises when detecting the similarities between samples. In future work, combining the dynamic analysis [145] with the static analysis is a promising way to reduce the side-effects caused by the dead code that will never be executed.

5) Matching of Abstract Behaviors: For CTDroid, in addition to the specific sensitive behaviors generated by using 14 semantic groups and a set of Android system related concepts, there are some abstract behaviors that we fail to accurately match them with the dalvik code. For example, we cannot detect whether a given app contains the abstract behavior “launchł¿root exploits”, since the presence of root exploits in malware relies on expected runtime environment (e.g., specific vulnerable device driver or preconditions) [68]. In future work, we plan to transform the abstract behaviors into a list of specific behaviors and then design more specific features to address the limitation of matching abstract behaviors.

Bibliography

[1] Security alert: New beanbot sms trojan discovered. https://www.csc2.ncsu.edu/faculty/xjiang4/BeanBot/, 2011.

[2] Mobile security alerts. https://www.csc2.ncsu.edu/faculty/xjiang4/alerts.html/, 2012.

[3] Droidbench benchmarks. https://github.com/secure-software-engineering/DroidBench, 2013.

[4] Checkpoint. https://blog.checkpoint.com/, 2017.

[5] Cheetahmobile. http://www.cmcm.com/blog/en/, 2017.

[6] Fortinet. https://blog.fortinet.com, 2017.

[7] Lookout. https://www.lookout.com/, 2017.

[8] Paloalto. https://www.paloaltonetworks.com/, 2017.

[9] Securelist. https://securelist.com, 2017.

[10] Securityintelligence. https://securityintelligence.com, 2017.

[11] Trendlabs. http://blog.trendmicro.com, 2017.

[12] Wikipedia. https://en.wikipedia.org/wiki/Main_Page, 2017.

[13] Android develper. https://developer.android.com/index.html, 2018.

[14] Anzhi market. http://www.anzhi.com/, 2018.

[15] Apk protect. https://sourceforge.net/projects/apkprotect, 2018.

[16] Apktool: A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/, 2018.

[17] Dasho. https://www.preemptive.com/products/dasho/overview, 2018.

[18] Dropping common terms: stop words. https://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html, 2018.

[19] Eset-nod32. https://www.eset.com/us/home/antivirus/, 2018.

[20] Google play. https://play.google.com/store, 2018.

[21] gtriescanner. http://www.dcc.fc.up.pt/gtries/, 2018.

[22] High-productivity software for complex networks. https://networkx.github.io/, 2018.

[23] jsoup: Java html parser. https://jsoup.org/, 2018.

[24] Mcafee. http://us.mcafeestore.com/, 2018.

[25] Mobile malware mini dump. http://contagiominidump.blogspot.hk/, 2018.

[26] Proguard. http://proguard.sourceforge.net/, 2018.

[27] The stanford parser: A statistical parser. https://nlp.stanford.edu/software/lex-parser.shtml, 2018.

[28] Virusshare. http://virusshare.com/, 2018.

[29] Virustotal. https://www.virustotal.com, 2018.

[30] Wordnet: A lexical database for english. https://wordnet.princeton.edu/wordnet/, 2018.

[31] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proc. SecureComm*, 2013.

[32] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine Learning*, 6(1), 1991.

[33] Annie I Anton and Julia B Earp. A requirements taxonomy for reducing web site privacy vulnerabilities. *Requirements Engineering*, 9(3):169–185, 2004.

[34] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[35] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. PLDI*, 2014.

[36] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proc. CCS*, 2012.

[37] Zarni Aung and Win Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.

[38] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. ICSE*, 2015.

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

[40] Smriti Bhagat, Graham Cormode, and S Muthukrishnan. Node classification in social networks. In *Proc. SNDA*, 2011.

[41] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proc. CCS*, 2016.

[42] Vincent Blondel, JeanLoup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008.

[43] Leo Breiman. Random forests. *Machine Learning*, 45(1), 2001.

[44] Gerardo Canfora, Andrea De Lorenzo, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Effectiveness of opcode ngrams for detection of multi family android malware. In *Proc. ARES*, 2015.

[45] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3), 2011.

[46] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proc. ICSE*, 2014.

[47] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proc. USENIX SEC*, 2015.

[48] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *Proc. NDSS*, 2013.

[49] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, (6), 2004.

[50] Giulio Concas, Cristina Monni, Matteo Orru, and Roberto Tonelli. A study of the community structure of a complex software network. In *Proc. WETSoM*, 2013.

[51] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proc. ESORICS*, 2012.

[52] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *Proc. ESORICS*, 2013.

[53] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *Proc. ICML*, 2016.

[54] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proc. PPREW*, 2014.

[55] Egham. Gartner says worldwide sales of smartphones returned to growth in first quarter of 2018. `https://www.gartner.com/newsroom/id/3876865`, 2018.

[56] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proc. CCS*, 2009.

[57] Pablo A Estévez, Michel Tesmer, Claudio A Perez, and Jacek M Zurada. Normalized mutual information feature selection. *IEEE TNN*, 20(2):189–201, 2009.

[58] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu. Frequent subgraph based familial classification of android malware. In *Proc. ISSRE*, 2016.

[59] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 13(8):1890–1905, 2018.

[60] Adrienne Porter Felt, Serge Egelman, and David Wagner. I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In *Proc. SPSM*, 2012.

[61] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proc. FSE*, 2014.

[62] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *Proc. NDSS*, 2017.

[63] Edward B Fowlkes and Colin L Mallows. A method for comparing two hierarchical clusterings. *Journal of the American statistical association*, 78(383):553–569, 1983.

[64] Eibe Frank and Ian H Witten. Generating accurate rule sets without global optimization. In *Proc. ICML*, 1998.

[65] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. S&P*, 2010.

[66] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[67] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proc. AiSec*, 2013.

[68] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V Krishnamurthy. Detecting android root exploits by learning from root providers. In *Proc. USENIX SEC*, 2017.

[69] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proc. ICSE*, 2014.

[70] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proc. MobiSys*, 2012.

[71] Vijayendra Grampurohit, Vijay Kumar, Sanjay Rawat, and Shatrunjay Rawat. Category based malware detection for android. In *Proc. SSCC*, 2014.

[72] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proc. KDD*, 2016.

[73] David Guthrie, Ben Allison, Wei Liu, Louise Guthrie, and Yorick Wilks. A closer look at skip-gram modelling. In *Proc. LREC*, 2006.

[74] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proc. DMIVA*, 2012.

[75] Robert C Holte. Very simple classification rules perform well on most commonly used datasets. *Machine learning*, 11(1):63–90, 1993.

[76] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *Proc. SIGKDD*, 2017.

[77] Wenjun Hu, Jing Tao, Xiaobo Ma, Wenyu Zhou, Shuang Zhao, and Ting Han. Migdroid: Detecting app-repackaging android malware via method invocation graph. In *Proc. ICCCN*, 2014.

[78] Qihoo Inc. Report of smartphone security in china, 2016 q3. `https://goo.gl/V9Vh1u`, 2016.

[79] Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based behavior analysis for android malware detection. In *Proc. CIS*, 2001.

[80] Adrian Johnstone, Elizabeth Scott, and Tim Womack. What assembly language programmers get up to: Control flow challenges in reverse compilation. In *Proc. CSMR*, 2000.

[81] Gordon Kelly. 97% of mobile malware is on android. this is the easy way you stay safe. `http://goo.gl/MYDBKC`, 2014.

[82] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4), 2011.

[83] Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

[85] Jin Li, Lichao Sun, Qiben Yan, Zhiqiang Li, Witawas Srisa-an, and Heng Ye. Significant permission identification for machine learning based android malware detection. *IEEE TII*, 14(7):3216–3225, 2018.

[86] Li Li, Tegawende Bissyande, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proc. ISSTA*, 2016.

[87] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *Proc. SANER*, 2016.

[88] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Proc. ICSE*, 2017.

[89] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the Association for Information Science and Technology*, 58(7):1019–1031, 2007.

[90] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers and Security*, 39:340–350, 2013.

[91] Xing Liu and Jiqiang Liu. A two-layered permission-based android malware detection scheme. In *Proc. Mobilecloud*, 2014.

[92] Zhang Luoshi, Niu Yan, Wu Xiao, Wang Zhaoguo, and Xue Yibo. A3: automatic analysis of android malware. In *Proc. IWCCIS*, 2013.

[93] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[94] Niccolò Marastoni, Andrea Continella, Davide Quarta, Stefano Zanero, and Mila Dalla Preda. Groupdroid: Automatically grouping mobile malware by extracting code similarities. In *Proc. SSPREW*, 2017.

[95] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proc. NDSS*, 2016.

[96] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proc. ISSTA*, 2016.

[97] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS*, 2013.

[98] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2014.

[99] Omid Mirzaei, Guillermo Suarez-Tangil, Juan Tapiador, and Jose M de Fuentes. Triflow: Triaging android applications using speculative information flows. In *Proc. AsiaCCS*, 2017.

[100] Veelasha Moonsamy, Jia Rong, and Shaowu Liu. Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*, 36:122–132, 2014.

[101] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[102] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2), 2004.

[103] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proc. KDD*, 2016.

[104] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proc. USENIX SEC*, 2013.

[105] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proc. KDD*, 2014.

[106] Robin L Plackett. Karl pearson and the chi-squared test. *International Statistical Review*, 51(1):59–72, 1983.

[107] Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Lidan Wang, Yuqiao Hou, and Zijiang Yang. Exploring community structure of software call graph and its applications in class cohesion measurement. *J SYST SOFTWARE*, 108:193–210, 2015.

[108] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proc. CCS*, 2014.

[109] Usha Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3), 2007.

[110] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

[111] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proc. NDSS*, 2014.

[112] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1), 2014.

[113] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. struc2vec: Learning node representations from structural identity. In *Proc. KDD*, 2017.

[114] Pedro Ribeiro and Fernando Silva. Querying subgraph sets with g-tries. In *Proc. DBSocial*, 2012.

[115] Kaspar Riesen, Sandro Emmenegger, and Horst Bunke. A novel software toolkit for graph edit distance computation. In *Proc. GbRPR*, 2013.

[116] Eric Sven Ristad and Peter N Yianilos. Learning string-edit distance. *IEEE TPAMI*, 20(5), 1998.

[117] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proc. EMNLP-CoNLL*, 2007.

[118] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *PNAS*, 105(4), 2008.

[119] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4):296–298, 1990.

[120] Steven L Salzberg. C4.5: Programs for machine learning. *Machine Learning*, 16(3), 1994.

[121] Borja Sanz, Igor Santos, Xabier Ugarte-Pedrero, Carlos Laorden, Javier Nieves, and Pablo García Bringas. Anomaly detection using string analysis for android malware detection. In *Proc. SOCO*, 2014.

[122] Sebastian Schuster and Christopher D Manning. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proc. LREC*, 2016.

[123] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proc. ACSAC*, 2014.

[124] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proc. IEEE S&P*, 2016.

[125] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In *Proc. ICSE*, 2016.

[126] Douglas Steinley. Properties of the hubert-arable adjusted rand index. *Psychological methods*, 9(3):386, 2004.

[127] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4), 2014.

[128] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *Proc. SEC*, 2014.

[129] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *Proc. IWRAID*, 2014.

[130] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). *Security and Communication Networks*, 9(5):389–419, 2016.

[131] Ke Tian, Danfeng Yao, Barbara G Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Proc. SPW*, 2016.

[132] Zhenzhou Tian, Ting Liu, Qinghua Zheng, Ming Fan, Eryue Zhuang, and Zijiang Yang. Exploiting thread-related system calls for plagiarism detection of multithreaded programs. *Journal of Systems and Software*, 119:136–148, 2016.

[133] Zhenzhou Tian, Qinghua Zheng, Ting Liu, Ming Fan, Eryue Zhuang, and Zijiang Yang. Software plagiarism detection with birthmarks based on dynamic key instruction sequences. *IEEE Transactions on Software Engineering*, 41(12):1217–1235, 2015.

[134] Julian R Ullmann. An algorithm for subgraph isomorphism. *JACM*, 23(1), 1976.

[135] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proc. ISSTA*, 2015.

[136] Wei Wang, Yuanyuan Li, Xing Wang, Jiqiang Liu, and Xiangliang Zhang. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. *Future Generation Computer Systems*, 78:987–994, 2018.

[137] Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Exploring permission-induced risk in android applications for malicious application detection. *IEEE Transactions on Information Forensics and Security*, 9(11):1869–1882, 2014.

[138] Xing Wang, Wei Wang, Yongzhong He, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. Characterizing android apps' behavior for effective detection of malapps at large scale. *Future Generation Computer Systems*, 75:30–45, 2017.

[139] Sebastian Wernicke and Florian Rasche. Fanmod: A tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[140] Britton Wolfe, Karim O Elish, and Danfeng Daphn Yao. Comprehensive behavior profiling for proactive android malware detection. In *Proc. ICISC*, 2014.

[141] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proc. Asia JCIS*, 2012.

[142] HoChung Wu, Robert Luk, KamFai Wong, and KuiLam Kwok. Interpreting tf-idf term weights as making relevance decisions. *ACM TOIS*, 2008.

[143] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proc. CCS*, 2017.

[144] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *Proc. ICSE*, 2017.

[145] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. Malton: Towards on-device non-invasive mobile malware analysis for art. In *Proc. USENIX SEC*, 2017.

[146] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proc. ESORICS*, 2014.

[147] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *Proc. DSN*, 2016.

[148] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proc. WiSec*, 2014.

[149] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. CCS*, 2014.

[150] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. CCS*, 2013.

[151] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Proc. ESORICS*, 2015.

[152] Min Zhao, Fangbin Ge, Tao Zhang, and Zhijian Yuan. Antimaldroid: An efficient svm-based malware detection framework for android. In *Proc. ICICA*, 2001.

[153] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proc. SPSM*, 2014.

[154] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proc. CODASPY*, 2013.

[155] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. CODASPY*, 2012.

[156] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE S&P*, 2012.

[157] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proc. CCS*, 2016.