Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON SECURITY AND PRIVACY FOR CLOUD AND IOT

Received April 22, 2019, accepted May 15, 2019, date of publication May 22, 2019, date of current version June 5, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2918139

Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions

WEI WANG[1], MEICHEN ZHAO[1], ZHENZHEN GAO[1], GUANGQUAN XU[2], HEQUN XIAN[3], YUANYUAN LI[1], AND XIANGLIANG ZHANG[4]

[1]Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing 100044, China
[2]Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, Tianjin 300350, China
[3]College of Computer Science and Technology, Qingdao University, Qingdao 266071, China
[4]Division of Computer, Electrical and Mathematical Sciences and Engineering, King Abdullah University of Science and Technology (KAUST), Thuwal 23955, Saudi Arabia

Corresponding author: Guangquan Xu (losin@tju.edu.cn)

The work was supported in part by the National Key R&D Program of China, under Grant 2017YFB0802805, and in part by the Natural Science Foundation of China under Grant U1736114.

ABSTRACT The number of applications (apps) available for smart devices or Android based IoT (Internet of Things) has surged dramatically over the past few years. Meanwhile, the volume of ill-designed or malicious apps (malapps) has been growing explosively. To ensure the quality and security of the apps in the markets, many approaches have been proposed in recent years to discriminate malapps from benign ones. Machine learning is usually utilized in classification process. Accurately characterizing apps' behaviors, or so-called features, directly affects the detection results with machine learning algorithms. Android apps evolve very fast. The size of current apps has become increasingly large and the behaviors of apps have become increasingly complicated. The extracting effective and representative features from apps is thus an ongoing challenge. Although many types of features have been extracted in existing work, to the best of our knowledge, no work has systematically surveyed the features constructed for detecting Android malapps. In this paper, we are motivated to provide a clear and comprehensive survey of the state-of-the-art work that detects malapps by characterizing behaviors of apps with various types of features. Through the designed criteria, we collect a total of 1947 papers in which 236 papers are used for the survey with four dimensions: the features extracted, the feature selection methods employed if any, the detection methods used, and the scale of evaluation performed. Based on our in-depth survey, we highlight the issues of exploring effective features from apps, provide the taxonomy of these features and indicate the future directions.

INDEX TERMS Android system, IoT, security and privacy, machine learning, malware analysis, malapp detection, survey.

I. INTRODUCTION

In recent years, Android has become the most popular mobile operating system. Millions of applications (apps) have been developed for Android smart devices. Meanwhile, the number of malicious applications (malapps) explosively increases. According to the 2018 Android Malware Special Report [1] given by 360 Internet Security Center, a total of 4.34 million new malware samples on Android platform, an average of 12 thousand per day, have been intercepted by 360 Internet Security Center in 2018. Many malwares were developed for smart devices or in IoT (Internet of Things). AppBrain [2] indicated that at the end of February 2019 the number of available apps on Google Play has reached over 2.5 millions, 12% of which are seen as low quality or potentially unwanted apps.

In order to keep malapps and massive low quality apps out of the markets, a number of malapp analysis and detection systems have been developed by analyzing the behaviors of apps. These behaviors are depicted by the apps' characteristics, a.k.a. features. The process of Android malapp detection is shown in Fig. 1. Clearly, machine learning is widely used

The associate editor coordinating the review of this manuscript and approving it for publication was Kuo-Hui Yeh.

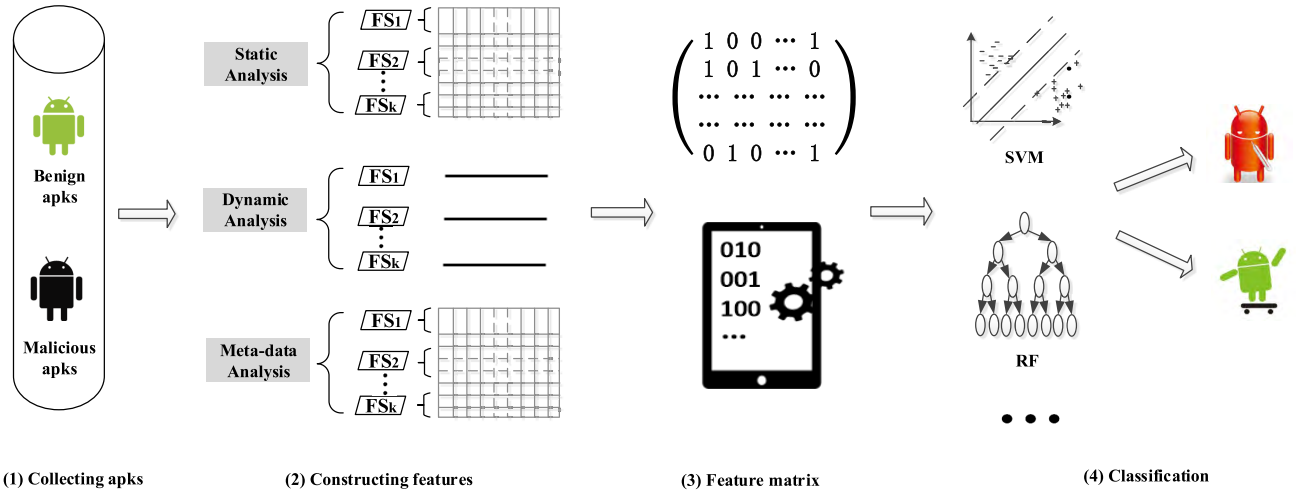


FIGURE 1. The process of Android malapp detection.

in classification process and constructing features is the most important step for Android malapp detection. The quality of the selected features determines the performance of the detection. Existing features can be classified into 3 categories, i.e., features extracted with static analysis (ab. static features), features extracted with dynamic analysis (ab. dynamic feature) and meta-data based features. Although the effort on the detection of malapps with extracted features from apps has made rapid progress, there still exists a number of issues.

A. ISSUES

1) COMMON ISSUES OF EXTRACTING THE THREE CATEGORIES OF FEATURES

We itemize the common and critical issues of extracting the three categories of features.

- The extraction of features may be time-consuming due to the increasing size and highly complicated behaviors of an Android Package (APK), resulting in the noneffective detection. For example, it often requires 15 minutes to extract function call graphs for an apk from Google Play with 15MB with static analysis. Clearly this is not acceptable for real-time detection for end-users.
- The number of features extracted from an app can be up to a million. However, many features are zero [3]. How to efficiently process the sparse vectors is an important issue.

2) ISSUES OF EXTRACTING STATIC FEATURES

Static analysis is widely used for vetting apps. However, there exist several key challenges that static analysis is facing.

- It is difficult to extract well-discriminated static features as the behaviors of Android apps have been increasingly polymorphic and sophisticated.
- The number of features rapidly increases with the increasing number of apps. Effective and efficient processing of the fast growing number of features is an important issue. In our previous work [3], we have categorized all the static features into two types: platform-defined features and app-specific features, according to the generality and specificity of feature sets. In terms of quantity, the app-specific features grow with the increase of app set, while the amount of platform-defined features keeps stable instead. The platform-defined features are more persistent than app-specific features, and thus they can be generally used for automated detection of malapps. Basically the number of app-specific features are quite large for processing, resulting in possible non-effective detection.
- Many malapps evade the detection based on static features through obfuscation techniques like dynamical loading code or code encryption. According to a large scale investigation [4], roughly 25% of apps in Google Play are obfuscated, and this number rises to 50% for the most popular apps with more than 10 million downloads. In general, static analysis techniques are easier to conduct and more effective than dynamic analysis; unfortunately, many static analysis techniques are easily thwarted by obfuscation.

3) ISSUES OF EXTRACTING DYNAMIC FEATURES

Dynamic analysis can extract inconspicuous behaviors of malapps. By monitoring and recording the behaviors of an app, the information observed may reflect the app's exact intention. However, there still remains some issues on extracting dynamic features.

- Dynamic analysis cannot traverse all possible execution paths, thus the malapp detection based on the dynamic feature may result in false negatives.
- Dynamic features may not be extracted if an app is protected by runtime security mechanisms (e.g., DexGuard).

Meta-data features have contributory values in judging the malicious behaviors of apps. However, the detection results

on malapps based on meta-data features only are far from accurate. In addition, the meta-data information of apps is easily out of date.

B. CONTRIBUTIONS

Definition and extraction of well-discriminated features is one of the most important components in effective detection of malapps. Although many types of features have been extracted in existing work, to the best of our knowledge, no work has systematically surveyed on the features constructed for the detection of malapps. In this work, we are motivated to provide a clear and comprehensive view of the state-of-the-art work that detects malapps by characterizing behaviors of apps with various features, from which we highlight the issues of exploring effective features, provide the taxonomy of features and indicate the future directions.

We make the following contributions.

- To the best of our knowledge, this is the first comprehensive and exhaustive survey on Android apps' features constructed for malapp detection in smart devices. We review thousands of published papers and finally select 236 of them, to form a survey of features extracted in existing work on Android malapp detection. We provide mainstream features that can be used for malapp detection, and accordingly summarize a taxonomy of all these features.
- We provide a clear view of the state-of-the-art work that vets Android apps. Based on our in-depth survey, we highlight the issues of defining and exploring well-discriminated features.
- We exhibit and describe the most used static features, dynamic features and meta-data features. We further summarize the feature selection methods used in existing work. We carry out a comparison among the related work, and point out the unique contribution of these work. We then classify the existing work according to the taxonomies of their employed features.
- We point out what should be focused on in future work. We conclude the trends of this research field and provide directions for future research.

The rest of this survey is organized as follows. we give a brief introduction to the characteristics of Android in Section II. Section III introduces the methodology of this survey. In Section IV, we survey the types of features used in malapp detection. Section V summarizes the feature selection methods used in existing research on malapp detection. Section VI provides related surveys and Section VII gives some discussions about reviewed papers and some promising directions for future research. The last section concludes this paper.

II. OVERVIEW OF ANDROID SYSTEM AND SECURITY

Before presenting our survey, we first provide a brief introduction to the Android platform and its incorporated security mechanisms. The fundamental information will help to understand the challenges and threats in Android platform. It's vital information for solving new problems raised by Android application security analysis methods and technologies.

A. ANDROID PLATFORM

Android is an open-source mobile operating system based on Linux kernel and designed primarily for smart devices. Android system has a hierarchical structure that consists of Linux kernel layer, library layer, application framework layer and application layer. Linux kernel layer provides some basic functions such as memory management, process management, and network protocols. This layer contains the core drivers for all underlying devices of the hardware components. Library layer provides the core library that includes the original library and third-party library for apps in order to assist application framework layer. The application framework layer is equivalent to an intermediate layer that intelligently coordinates the components, which enhances the flexibility of the entire system. To complete this work, application framework contains many system services such as Activity Manager, Window Manager, Resource Manager, Location Manager, Content Provider and so on. The application layer, the only layer that can interact with users, consists of all apps running on Android devices. There exist solutions with secure protocols for IoT security [5], [6]. However, in this work, we focus on application security in Android system deployed in smart devices.

B. ANDROID APPLICATIONS

An Android app is written in Java programming language using APIs provided by the Android Software Development Kit (SDK). Besides the Java code, an app may also contain some native libraries that are provided by Android system or implemented by developers. An app's compiled code alongside data and resources are packed into an archive file, known as an Android Application Package (APK). Once an APK is installed on an Android device, it runs by using the Android runtime environment.

Android apps contain four main components: Activity, Broadcast Receivers, Service and Content Provider. Activity dictates the User Interface and handles the user interaction with the smart phone screen. Broadcast Receivers deal with communication between operating system and apps. Service manages background processing of an app to perform long-running operations. Content Provider provides the data sharing across apps.

C. INCORPORATED SECURITY MECHANISMS

Developers introduce various security mechanisms when they design the Android platform. Android system has a hierarchical structure, and each layer has its own security mechanism, namely, traditional access control mechanism, mechanism based on inspection of permission, sandbox mechanism, digital signature mechanism and encryption mechanism.

1) TRADITIONAL ACCESS CONTROL MECHANISM

Traditional access control mechanism is equivalent to Android's Linux kernel security mechanism. Access control restricts the subject (such as users or services) to access the object (such as resources). It is a primary approach to protect the confidentiality and integrity of data. Access control includes two kinds of methods, mandatory access control (MAC) and discretionary access control (DAC). MAC is implemented by the Linux security module. DAC is implemented by file access control.

2) MECHANISM BASED ON INSPECTION OF PERMISSION

Android uses a permission-based security model to restrict apps access some resources. If apps want to use restricted resources, they have to apply for permissions through XML files. Apps cannot use restricted resources until Android system approves. Android permissions have four levels, namely Normal, Dangerous, Signature, and Signature/System. Low-level permissions, including normal and dangerous levels, are authorized as soon as an app applies for. Signature level and signature/system level permissions are known as advanced permissions. Before an app can apply these permissions, it needs to achieve platform-level authentication. However, there are many shortcomings in this mechanism. Users need to decide if the permissions that an app applies should be authorized, yet users do not have enough knowledge to judge it. Moreover, if the device is running on Android 5.1.1 (API level 22) or lower, or the app's target Sdk Version is lower than 23, the system automatically asks the user to grant all dangerous permissions for the app at install-time. once the permissions are authorized, they will remain valid for the duration of the app, unless users change.

3) SANDBOX MECHANISM

Sandbox is used to separate running apps in Android system. A sandbox provides a tightly controlled set of resources for apps to run in. During the run-time of the Android apps, each app runs in its own Dalvik virtual machine and has its own process space and resources. Therefore, different apps cannot interact with each other and cannot access each others' resources and memory space.

4) DIGITAL SIGNATURE MECHANISM

Digital signature mechanism plays a very important role in the security of application layer. Android app developers have to give their apps digital signatures, since the apps without digital signatures are not allowed to be installed. If an attacker deliberately changes the internal file of APK, he has to re-sign the app. Not until the attacker gets the private key of original publisher will he generate the signature that is consistent to the original signature. In addition, the signatures of apps will also be checked when apps need to be updated. Digital signature ensures the integrity and reliability of apps.

5) ENCRYPTION MECHANISM

Android system can support encryption mechanism that is to protect some important data from being accessed by unauthorized users or apps. Android system implements encryption mechanism in version 3.0 and above. Because users pay more and more attention to protecting private data, such as phone event, SMS and some payment information, encryption mechanism becomes increasingly important to Android system.

III. METHODS OF THE SURVEY

Our survey follows the general guidelines for Systematic Literature Review (SLR) process proposed by Sadeghi et al. [7]. We also apply SLR to static analysis, taking into account the lessons from Li et al. [8]. The whole process includes three main phases: planning, conducting, and reporting the reviews. Based on the guidelines, we formulate the following research questions that serve as the basis for the SLR.

RQ1: How can existing research work on Android app security analysis be classified?

RQ2: What features have been used for Android malapp detection?

RQ3: What methods have been used for feature selection?

A. RESEARCH TASKS

To answer the research questions introduced above, we organize our tasks into three-phase SLR process, as mentioned, including: planning the review, conducting the review, and reporting the review.

In the planning phase, we define the review protocols that include selection of the search engines, the initial selection of the keywords related to Android malapp detection, and the selection standard for the candidate papers. The protocols are described in detail in part B.

The literature selection is an iterative process that includes selecting candidate papers as well as applying the pre-defined inclusion/exclusion standards. In this process, the keyword search expressions and the inclusion/exclusion standards may also need to be adjusted, which would trigger new searches. Once the review protocols and the paper collection were finished, we read literature carefully to validate the selections.

For RQ1, in order to define a comprehensive categorization that is suitable for Android app security analysis, we first start with a quick survey on related reviews to summarize rough definitions of categorization. We then focus on abstract, introduction, contribution, and conclusion sections of papers to identify new concepts and approaches to augment and refine the categorization. The final categorization is presented in Section IV.

For the second research question (RQ2), we use the validated papers and the categorization of Android app security analysis to conduct a more detailed review of these papers. We conduct peer-reviews on papers in order to identify the features for malapp detection. The definition of features and

TABLE 1. Research domain and their keywords.

Research Domain (D)	Keywords (K)
Program Analysis	Static (Analysis), Dynamic (Analysis), Meta-data, Feature
Security Assessment	Security, Malicious (APP), Malware, Anomaly, Detection, Categorization, Feature Selection
Android Platform	Android, Mobile, Smartphone, Application

the reason why they are applied in malapp detection are also the content we explore. We judge which categorization the features belong to according to the papers. The analysis and findings are documented in Section V.

To answer the third research question (RQ3), we attempt to identify the feature selection approaches that some work applies in order to improve the efficiency and accuracy. The principle of the methods is also the content we explore. Section VI concludes the analysis and findings.

B. LITERATURE REVIEW METHODS

1) SEARCH METHOD

We use well-known literature search engines and databases to prepare our survey. Our goal is to find high-quality and high-correlation refereed papers, including journal articles and conference papers. The selected search engines consist of IEEE Explore, ACM Digital Library, Springer, Elsevier and DBLP. The journals and conferences are related to software engineering (including software testing and analysis), programming language and systems, security, and mobile computing and systems.

According to the topic of this survey, we focus on the keywords to perform the literature search, which are given in Table 1. Besides, we extend them with regular expressions and part-of-speech conversion to obtain more keywords.

2) SELECTION STANDARD

Not all the candidate papers based on the search method fit within the scope of this paper. Therefore, we use the following inclusion and exclusion standard to further filter the candidate papers.

a: INCLUSION STANDARD

As shown in Table 1, there are three domains that are composed of keywords. The topic of this survey falls at the intersection of three domains. Papers that fall at the intersection of these three domains can be included in our review.

1) Program Analysis domain covers the approaches and tools that are used for conducting Android app security analysis.

2) Security Assessment domain includes the expected targets that can be achieved through a series of experiments.

3) Android Platform domain focuses on the platform that the malapp detection conducts on.

FIGURE 2. Word cloud of all venue names of the collected papers.

b: EXCLUSION STANDARD

Moreover, we excluded:

1) Papers focus on platforms other than Android, such as iOS, Windows Mobile, BlackBerry, and Symbian. However, papers that cover multiple platforms, including Android, are included.

2) Papers focus only on techniques for mitigation of security threats, but not on any security analysis technique. Such techniques attempt to enhance security mechanisms either at the application-level or the level of the Android platform through different approaches.

3) Papers perform a particular attack on the Android framework or apps, without describing detection techniques or specific features used in detection.

4) Papers focus on app security analysis, without mentioning any type of features.

C. SELECTED PAPERS

We firstly collect 1947 research papers through the search method. Based on the selection standards and reviews on these papers, we finally select 236 papers. We perform some statistical analysis on these papers to figure out the distributions of them in the publication venues and the prevailing trend of each year. Fig. 2 and Fig. 3 describe the distribution of selected papers in each journal/conference, indicating that most selected papers are published in the top-tier conferences and journals. Fig. 4 shows the number of selected papers in each year from 2011 to 2019. In order to achieve more advanced review results, we mainly collect published papers up to date. From this figure, we can see that the number of papers about security analysis of Android apps is in general on the rise in recent years.

IV. TAXONOMY OF FEATURES CONSTRUCTED FOR ANDROID MALAPP DETECTION

As mentioned above, the number of malapps on Android smart devices has surged dramatically over the past years. In order to ensure the quality and security of apps in the markets, a number of systems have been developed to detect

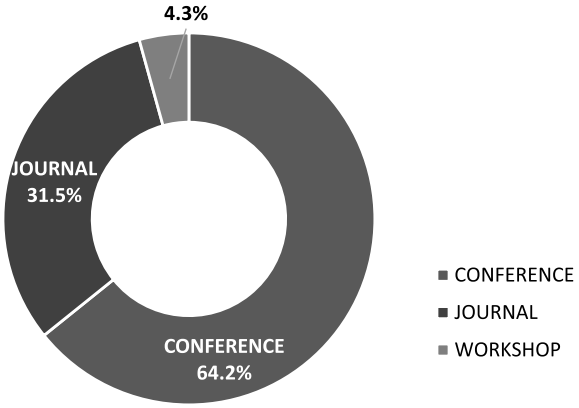


FIGURE 3. Statistic of types of the collected papers' publications.

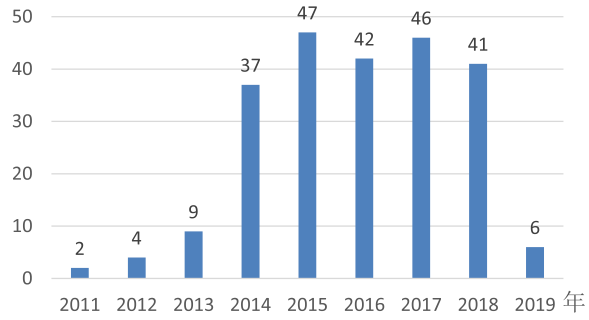


FIGURE 4. The numbers of selected papers in each year.

malapps. Malapp detection refers to separate malapps from benign ones through analyzing the behaviors of Android apps. Currently, malapp detection can be implemented through static analysis, dynamic analysis, hybrid analysis and meta-data analysis. Correspondingly, there are three types of features: static features, dynamic features, as well as meta-data features. The various types of features and sub-types of each category are presented in Fig. 5.

In this section, we present our taxonomy of features constructed for Android malapp detection. We first describe the analysis methods of detecting Android malapps. Then we exhibit and describe the most used static features, dynamic features and meta-data features. We further carry out a comparison among the related work, and point out the unique contributions of these work.

A. METHODS OF DETECTING ANDROID APPS

Existing analysis methods for detecting Android apps mainly consist of static, dynamic, hybrid and meta-data analysis. We introduce these analysis methods briefly and classify the surveyed papers according to the taxonomies of their employed features. Table 2 shows the number of these papers in each group.

1) STATIC ANALYSIS (124/236)

Because Android app has surged dramatically, Android platform becomes the target of attackers and suffers from

TABLE 2. the number of reviewed papers in each group.

Types of Features	Paper	Number
Only Static Features	[9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79], [80], [81], [82], [83], [84], [85], [86], [87], [88], [89], [90], [91], [92], [93], [94], [95], [96], [97], [98], [99], [100], [101], [102], [103], [104], [105], [106], [107], [108], [109], [110], [111], [112], [113], [114], [115], [116], [117], [118], [119], [120], [121], [122], [123], [124], [125], [126]	118
Only Dynamic Features	[127], [128], [129], [130], [131], [132], [133], [134], [135], [136], [137], [138], [139], [140], [141], [142], [143], [144], [145], [146], [147], [148], [149], [150], [151], [152], [153], [154], [155], [156], [157], [158], [159], [160], [161], [162], [163], [164], [165], [166], [167], [168], [169], [170], [171], [172], [173], [174], [175], [176], [177], [178], [179], [180], [181], [182], [183], [184], [185], [186]	60
Only Meta-data Features	[187], [188], [189], [190], [191], [192]	6
Static & Dynamic Features	[7], [193], [194], [195], [196], [197], [198], [199], [200], [201], [202], [203], [204], [205], [206], [207], [208], [209], [210], [211], [212], [213], [214], [215], [216], [217], [218], [219], [220], [221], [222], [223], [224], [225], [226], [227], [228], [229], [230], [231], [232]	41
Static & Meta-data Features	[3], [233], [234], [235], [236], [237]	6
Static & Dynamic & Meta-data Features	[238], [239], [240], [241], [242]	5

increasingly serious malapp threats. In response, much research work aims to detect malapps via static analysis. In static analysis, apps are firstly unpacked and decompiled into files that present essential information about the apps. Then these files are inspected that if there are malicious codes. Static analysis is well known in traditional malapp detection, gaining popularity as efficient mechanisms for market protection. It is useful for resource constrained Android devices as the analysis is performed without executing the app. Static analysis consumes much less resources and time. It is thus a relatively fast method. However, this approach can be thwarted by malapps that use reverse engineering techniques, such as obfuscation and repackaging.

2) DYNAMIC ANALYSIS (60/236)

In contrast, dynamic analysis seeks to identify malicious behaviors after deploying and executing the apps on the emulators or real devices. It generates snapshots of processor execution, network activity, system calls, SMS sent, phone calls, etc. to discriminate malapps from normal ones. This technique requires some human or automated interaction with apps, as malicious behaviors are sometimes triggered only after certain events occur. The information observed through dynamic analysis correctly reflects the app's actual execution. However, the dynamic analysis execution leads to excessive

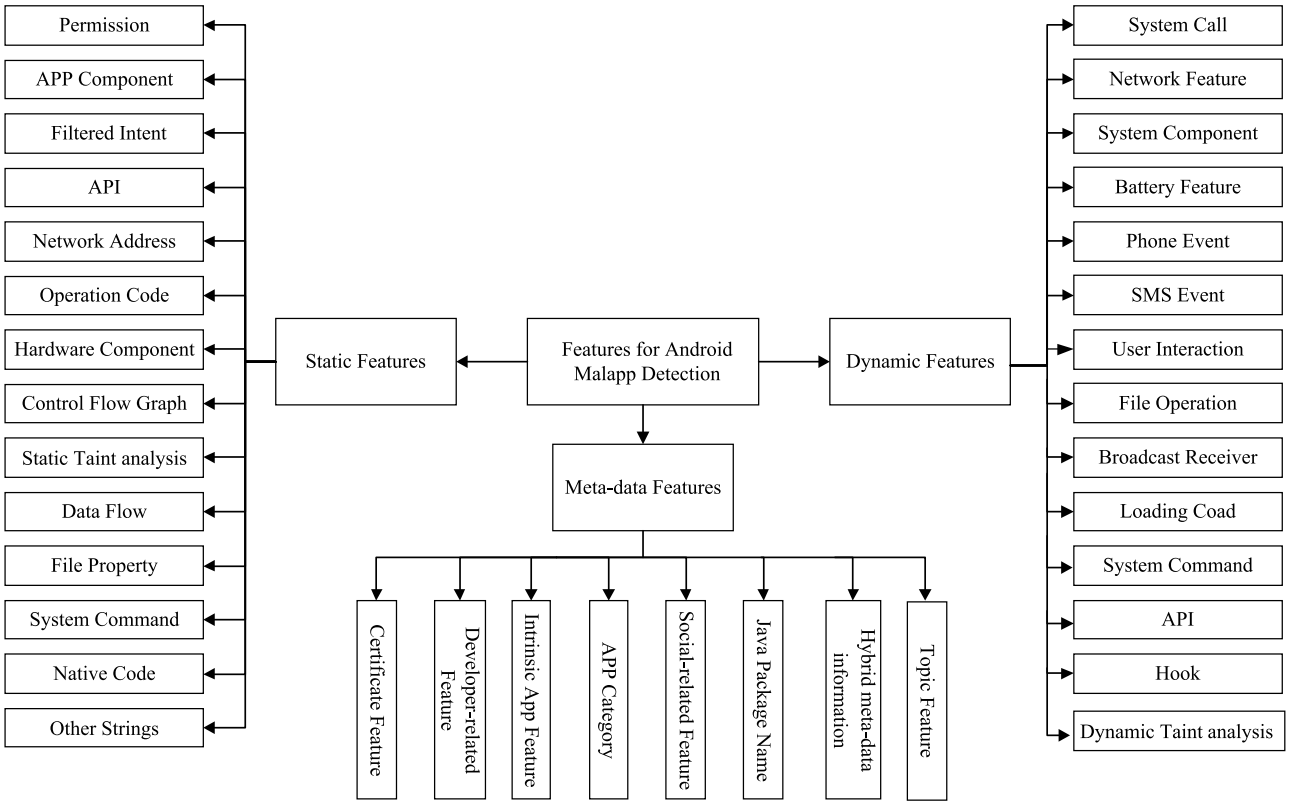


FIGURE 5. Various types of features and subtypes of each category.

consumption of Android operating system (OS) and time. Moreover, dynamic analysis method may not detect malapps that prevent themselves from running on the emulators.

3) HYBRID ANALYSIS (46/236)

We define hybrid analysis as using both static analysis, dynamic analysis and sometimes meta-data analysis in detection system. Hence, it combines the advantages and disadvantages of static analysis and dynamic analysis. It can be seen as the most comprehensive analysis because it analyzes both Android application installation files and behaviors of the app at runtime. The disadvantage of hybrid analysis is that it leads to excessive consumption of Android OS as well as wastes a lot of time.

4) META-DATA ANALYSIS (6/236)

We define meta-data as the information users see before downloading or installing apps. Meta-data analysis is a kind of indirect app analysis to identify information that is often observed in malapps. Meta-data analysis cannot be classified as static analysis or dynamic analysis since it has nothing to do with apps themselves. Some related work detects malapps through meta-data analysis, such as the category information of apps, the version of apps and the serial numbers of apps' certificates.

As illustrated in Table 2, the research on Android malapp detection still focuses on static analysis in recent years. The use of dynamic analysis method is in the rising period. Although the hybrid analysis is more comprehensive, they constitute only 18.8% of the literature. The obvious reason is that the hybrid analysis requires collecting both static and dynamic and even meta-data features. Using two or three types of features is a complicated process and costs excessive Android OS and time. In the next sections, we exhibit and describe the most used static features, dynamic features and meta-data features in detail. Additionally, we point out the unique contribution of the related work.

B. STATIC FEATURES

Static features can be extracted by analyzing the disassembly code of apps without executing them. Static features include features available in the apk files such as *AndroidManifest.xml* file and Java code file. Out of 234 papers reviewed, 168 papers use static features to conduct their experiments. The number of papers that uses various of static features are shown in Fig. 6. Among the static features, Android permissions are used in 71 papers, more than the use frequency of other static features. The accuracy of permissions-based detection achieves around 90% [11], [29], [37], [41], which is improved with additional features [3], [216], [233]. The following sections discuss these static features in details.

1) PERMISSION

In order to protect users' information security, Android uses a permission-based security model to restrict apps from

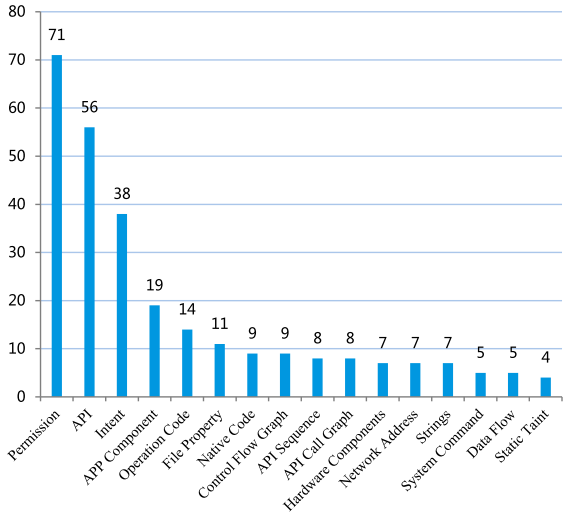


FIGURE 6. The numbers of papers using various of static features.

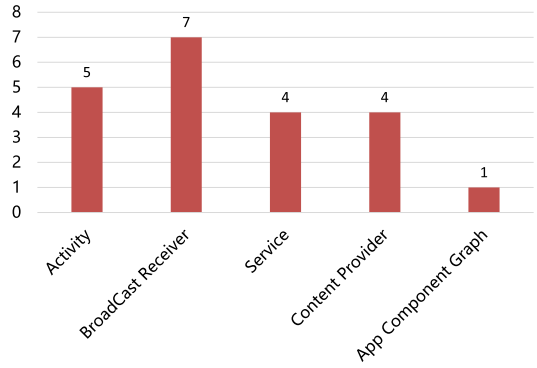


FIGURE 7. The frequency of each component used in malapp detection.

accessing users' sensitive information. An app provides a list of requested permissions to the users before it is installed. After these permissions are granted by users, the app installs itself on the device. In Android system, there are many kinds of permissions can be queried through the SDK documents. The combination of multiple permissions may reflect some harmful behaviors. For example, if an app applies for network connecting permission as well as SMS accessing permission, the app may acquire users' SMS information and then spread it out through the Internet.

There exist many approaches for detecting Android malapps by extracting permissions. Wang et al. [11] analyzed the risks of individual permissions and collaborative permissions. They ranked the individual permissions with respect to their risks. Sarma et al. [22] used both the permissions that ap app requested and permissions requested by other apps in the same category. The purpose of this method was to verify whether the app's benefits outweighed its expected risks. Some studies [3], [29], [37]–[39], [41], [44], [46], [77], [83], [196], [214], [216], [233], [235], [238] extracted permissions as well as some other features and utilized machine learning to detect malapps. This approach usually achieved accuracy as more than 94%. Liu and Liu [27] employed the requested permission for malapp detection. While Lindorfer et al. [235] and Wang et al. [3] chose not only the requested permissions, but also permissions based on the app's API calls, which was so-called used permissions. Kang et al. [230] excluded some permissions. For instance, permission INTERNET was excluded since it was required for most apps. On the contrary, INSTALL PACKAGES, a permission usually used to install a new package, was included instead. Some studies [43], [62], [67] considered permission combinations as the feature. Chen et al. [88] explored the security of machine learning in Android malapp detection. They integrated their proposed feature selection method (named SecCLS) and ensemble learning approach (named SecENS) to enhance security of machine learning-based Android malapp detection.

Android permission is the most used and effective static feature. It is because applying for permission is crucial for attackers to achieve their malicious goals. Although the Java code contains the implementation of malicious methods, some API calls requires permissions' permits. For example, before sending a SMS, Android system will check if the corresponding permissions are granted. Based on such scenario, existing work thus pays more attention to permissions than other static features in malapp detection.

2) APP COMPONENT

App components are the basic building blocks of an Android app. They are the entry points for system to access apps. Each component exists as a distinct entity and plays a specific role. These components are linked by the app's manifest file *AndroidManifest.xml*, which describes every component and how they interact. There are four main components within an Android app: Activity, Service, Broadcast Receiver, and Content Provider.

An activity dictates the User Interface (UI) and handles the user interaction with the smart phone screen. If an app has more than one activity, then one of them should be marked as the activity that is presented when the app launched. A service manages app's background process to perform long-running operations. For instance, a service might play music in the background while the user is in another app. Broadcast Receiver deals with communication between operating system and apps. For example, apps can inform other apps using broadcasts that some data has been downloaded and is available for them. A content provider resolves data and database management issues. The data can be stored in the file system, database or somewhere else. There are additional components that are used in the construction of above mentioned entities, such as Fragments, Views, Layouts, Resources, Manifest and so on.

There exists much work utilizing app components for Android malapp detection. Fig. 7 shows the frequency of each component used in malapp detection. Some studies [19], [34], [69], [184], [195] considered activity as features in malapp detection. Shao et al. [19] extracted the number of activities and other features to detect malapps. Studies [34], [69], [195] further applied service and broadcast receiver

as features in malapp detection. Feldman et al. [231] chose the frequency of high priority receivers and abused services to detect malapps. The accuracy was up to 90%, while the FPR and FNR were both around 10%. Mohsen et al. [92] analyzed the Java code and investigated the usage patterns of the Broadcast receiver components of malicious and benign Android apps. The experiments showed that using the Broadcast receivers with permissions increased the malapps' prediction accuracy to 97%. Studies [15], [18], [69] extracted content provider and other features and utilized machine learning for malapp detection. Varsha et al. [69] achieved accuracy of 98.14% with F-measure of 0.976 using Random Forest (RF) classifier. Shen et al. [26] studied topology graph based on app components. This method could model malicious payloads properly and resist against common obfuscation used by hackers. The evaluation result was that 86.36% of obfuscated malapps were caught with tolerable false positive. Different from previous work, Bosu et al. [95] developed a scalable and accurate tool DIALDroid for inter-app Inter-Component Communication (ICC) analysis. They performed the first large-scale detection for collusive and vulnerable apps based on inter-app ICC data flows. Liu et al. [213] presented MR-Droid, a Map Reduce-based computing framework for accurate and scalable inter-app ICC analysis on Android. Alatwi et al. [83] proposed category-based machine learning classifier to enhance the performance of classification models under a certain category. They utilized broadcast receivers as one of the features for malapp detection, and compared them with the Android broadcast events. Rehman et al. [217] proposed framework considering both signature and heuristic-based analysis for detecting Android apps. During static analysis, they extracted some representative features including providers' and receivers' information. And then, they evaluated the effectiveness of this framework utilizing multiple machine learning algorithms.

3) FILTERED INTENT

Intent message handles the communication between components of the same or different apps by sending intent objects. In order to inform Android system which intents can be received by the app component, each of them has one or more intent filters. Intent filters help app components reject the unwanted intents and leave the desired intents. They are described in the manifest files and have been used in malapp detection.

Lindorfer et al. [235] extracted the intents that app responded to through the broadcast receiver. Arp et al. [33] collected static features from Android installation file including filtered intents. They used SVM for detection purpose and the experimental results showed that DREBIN detected 94% of malapps with low false alarm. Xu et al. [87] developed a static analysis tool named AppHolmes for detecting app collusion by examining the app binaries. They extracted many static features including explicit and implicit intents. They presented an in-depth study of app collusion. Xu et al. [10] used explicit intents and implicit intents to conduct experiments and got the TPR of 93.1% with FPR of 0.67%. Feizollah et al. [76] also evaluated the effectiveness of Android Intents (explicit and implicit). They considered it as a distinguishing feature for identifying malapps. They also conducted experiments using Android Intent in conjunction with permission, resulting in detection rate of 95.5%. Besides permission combination, Song et al. [43] extracted 15 types of dangerous intents. They detected 4,006 real malapps and their accuracy rates were nearly 99%. Idrees et al. [78] used a combination of permissions and intents for identifying Android malapps. They optimized the results with ensemble methods furthermore, resulting in 99.8% accuracy.

4) API

API calls present how an app interacts with the Android framework. Every Android app needs API calls to interact with the device. Thus some work employs API calls as features for malapp detection. It is essential to capture the API calls and the dependencies among these calls. These information can be acquired through both static analysis and dynamic analysis.

There are different approaches for Android malapp detection by analyzing apps' API. Some studies [3], [14], [19], [35], [42], [51], [55], [66], [67], [83], [96], [150], [194], [197], [198], [204], [230], [231], [233] extracted APIs as well as some other features and utilized machine learning to detect malapps. Some studies employed the APIs under certain conditions. Lindorfer et al. [235] and Yerima et al. [55] used cryptographic API and reflection API as features. Besides, Yerima et al. utilized the APIs for SMS, telephony, JNI usage, dynamic class loading, the creation of new processes and the runtime execution of processes. Studies [3], [33], [46], [194] all considered restricted API calls and suspicious API calls as features to detect malapps. Instead of using API calls directly, Hou et al. [82] further categorized the API calls belonging to the same method in the smali code into a block, namely API call block. Their experimental results showed that the API call block outperformed using API calls directly in Android malapp detection. Wu et al. [49] chose the dataflow-related APIs as features. Saracino et al. [184] extracted operational APIs that might be critical for Android app security analysis, such as the APIs for installing a new app, requesting administrator privileges, generating too many processes and constantly monitoring the app. Maiorca et al. [94] leveraged information extracted from system API packages for detecting malapps. Shabtai et al. [57] extracted APIs for accessing data and APIs related to network.

There is a certain sequence for the API calls in a method. It's a strategy for attackers to change the API sequence in order to bypass the detection process, called code obfuscation. Therefore, some work considers API sequence as the app's unique feature to detect malapps. Studies [42], [47], [53], [81], [192], [196], [236] applied API sequence to detect malapps. Studies [53], [196] employed both API and API sequence as features. Meng et al. [53] used the bigram of

API sequence, i.e. 147 APIs related to Java reflection and 2 APIs used to invoke native code. Then they performed a series of experiments with different machine learning methods, achieving the highest accuracy of 97% with RF classifier.

API call graph can also reflect the dependencies among these API calls. In API call graph, each node represents an API and each edge (f, g) indicates that procedure f calls procedure g. Some work employs API call graph as the app's characteristic feature for malapp detection. Fig. 6 shows the usage frequency of API and API call graph in malapp detection.

Studies [17], [24], [32], [48], [84], [192], [202] employed API call graph in malapp detection. Dam et al. [84] applied well-known learning techniques based on Random Walk Graph Kernel combined with SVM, achieving a high detection rate as 98.76% with only 0.24% false alarms. Zhang et al. [17] introduced graph similarity metrics to uncover homogeneous apps' behaviors while tolerating minor implementation differences. The Experiments showed that this method correctly labeled 93% of malware instances. It was capable of detecting zero-day malapps with a low false negative rate (FNR) (2%) and an acceptable false positive rate (FPR) (5.15%). Because the connections between the injected malicious code and legitimate apps were expected to be weak, Hu et al. [24] used method invocation graph based on static analysis to detect repackaged Android malapps. It reflected the 'interaction' connections between different methods. The experimental results showed that the proposed method got the detection accuracy as 95.94%. Zhou et al. [192] employed API, API sequence, API call graph and other features for malapp detection.

5) NETWORK ADDRESS

Attackers need to contact malapps to report their status and send users' personal data. Therefore, attackers often add network address of the server, namely command & control (C&C) server, in malicious code. In malapp detection, some work looks for network address or IP address of the C&C server in app's installation files.

Studies [38], [59], [130] chose the URLs as one of the static features in their systems. Besides, Zhao et al. [38] extracted IP address as features. They used SVM and KNN for detection purpose and found that KNN was much faster than SVM with only 1%-2% decrease of accuracy. The results showed that FEST got nearly 98% accuracy and recall, with only 2% false alarms and high efficiency.

6) OPCODE

An opcode, standing for 'Operation Code', is a single instruction that can be executed by the CPU. The frequency of single opcode or opcode sequence from the apps can be seen as features in malapp detection. Some work focuses on opcodes because they are closely related to the app code.

Several studies showed that opcode could discriminate malapps from benign ones. In order to detect variants of known malapps, Hang et al. [71] extracted a simplified-instruction-set by static analysis on 218 Dalvik instructions. The experimental results showed that the proposed method can achieve a high hit rate with low FPR, and was more efficient than ordinary anti-virus software. Canfora et al. [40] investigated if n-grams frequencies of opcodes were effective in detecting Android malapps. Using both SVM and RF classifier, they evaluated it on a dataset that consist of 11,120 apps, 5560 of which were malapps from different families. The method achieved the highest accuracy when n equals 2. Puerta et al. [61] used the frequency of opcodes as features and found some particular opcodes. For example, an opcode called RSUB_INT was found only on benign apps. Hahn et al. [59] collected 14 types of static features including both frequency of opcodes and opcode sequences. They used RF, Bayesian Network, and KNN for detection purpose. The experimental results showed that, detection based on frequency of opcodes achieved accuracy as 94.82% with FPR of 1.25%. Detection based on opcode sequence got the same accuracy but with lower FPR, 0.65%. Mclaughlin et al. [80] proposed a malapp detection method that used a deep convolutional neural network (CNN). This system extracted opcode sequence from apps' Dalvik bytecode for static analysis. Its experiments were carried out on three different datasets. The results showed that this system was more efficient on a large scale dataset than n-gram based system. Ali et al. [85] scored similarity of opcode sequence found in sensitive functional modules, and used requested permissions to improve detection accuracy. The empirical results showed that their method can detect known malapps correctly with an F-Score of 98%. Martinelli et al. [237] proposed BRIDEMAID which combined static and dynamic analysis to detect Android malapps. This framework exploited the frequency of opcode and utilized SVM for detection.

7) HARDWARE COMPONENT

An app requests multiple hardware components to make app's function comprehensive. To some extent, the combination of requested hardware components can reflect harmful behaviors. For instance, if an app accesses 4G and GPS, this may imply that it can report user's location to the attacker.

Some work [33], [46], [194] showed that the combination of hardware components could discriminate malapps from benign ones. Shabtai et al. [191] collected both static features and dynamic features including hardware components. They focused on the camera hardware and the state of USB. They applied machine learning methods to classify the collected apps.

8) CFG

The control flow graph (CFG) is essential to static analysis. A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. Each node in a CFG corresponds to a basic block in the method. A basic block is a straight-line piece of code without any jumps or jump targets. Jump targets start a module, and

jumps end a module. Directed edges are used to represent jumps in the CFG. The code could be source code like JAVA, assembling code like SMALI, machine code like arm instructions or bytecode like DEX (Dalvik Executable). It's an approach for code obfuscation detection to analyze the control flow of Java code. Although attackers can change the sequence of API calls or rename API calls to evade detection, the flow of the Java code does not change.

Chen et al. [20] used CFG to measure the similarity of two apps. Then they synthesized the method-level similarities and predicted a conclusion on app cloning. Allix et al. [28] took static analysis on Android apps' bytecode to extract a representation of the program CFG. The extracted CFG is expressed as character strings. They used a dataset of over 50,000 Android apps. The system exhibited a very high precision rate with a median value of 0.94. Annamalai et al. [73] extracted the inter-procedural control flow sub-graph features for malapp detection. This system used online passive aggressive classifier and achieved 84.29% accuracy. Alam et al. [79] used control flow with patterns, and implemented and adapted two techniques including Annotated Control Flow Graph (ACFG) to reduce the effect of obfuscation. Dam et al. [84] constructed API call graph from the CFG by applying a kind of control point reachability analysis on the CFG, to carry out further experiments. Rehman et al. [217] utilized malware evolution attack and malware confusion attack to enhance feasibility of the attacks. They applied these two strategies for their presented Malware Recomposition Variation (MRV). They constructed inter-procedure control-flow graph (ICFG) to extract some features. They evaluated MRV on a dataset consisting of 1935 benign apps and 1917 malapps. The experimental results showed that MRV can have high likelihood to evade detection.

9) STATIC TAINT ANALYSIS

Taint analysis can be seen as a form of Information Flow analysis. Data flows from untrusted sources could cause security vulnerabilities in programs. Therefore, taint analysis can decide whether a leak actually constitutes a policy violation and it analyzes apps and presents potentially malicious data flows to human analysts or to automated malware-detection tools. Taint analysis includes static analysis and dynamic analysis. Static taint analysis keeps track of information flows by examining the source code (i.e. white-box testing). Static taint analysis provides better code coverage than dynamic analysis.

In 2014, Steven et al. [89] proposed a novel and highly precise static taint-analysis system, FlowDroid. It precisely modeled the complete Android lifecycle, including the correct handling of callbacks and user-defined UI widgets within the apps. Meanwhile Steven et al. developed an Android-specific test suite called DROIDBENCH. DROIDBENCH can be used to assess both static and dynamic taint analyses. Several research groups already used it to measure and improve the effectiveness of their Android analytics tools. In 2015, Chen et al. [90] extended Soot [239], Heros [240] and FlowDroid based on DroidJust to provide inter-procedural data flow analysis. They took a slightly different angle to tackle this privacy leakage detection problem. They proposed a novel idea that the information an app grep was reasonable and safe when the information was used to change the state of the Android device. DROIDJust used various static taint analyses to automate the whole analysis process. Their method can effectively and efficiently analyze both benign apps and malapps. In 2017, Mumtaz et al. [91] presented a rigorous literature review on most recent studies on static taint analysis. They mentioned that the most widely used techniques for performing taint analysis were Call Graph and Control Flow Graph. They then selected FlowDroid to conduct their experiments. The results showed that FlowDroid can successfully perform inter-component taint analysis with a marginally lower precision rate than intra-component communications.

10) DATA FLOW

Data flow analysis is to track the data across apps. It relies on an underlying abstract semantics of Android apps. Data Flow shows the data dependencies between functions. Based on data flow, security-relevant can decision automatically. Data Flow Graph (DFG) is a graphical representation of the 'flow' of data through an information system, modelling its process aspects. A DFG can present what kind of information will be input to and output from the system, where the data comes and goes, and where the data will be stored.

Zhou et al. [192] extracted many static features including data flow. They utilized data flow analysis algorithm to detect function parameters with static or fixed inputs. Based on the collected 204,040 apps, the system detected 211 infected apps and uncovered two zero-day malware, including one from the official Android Market. Amandroid [23] used inter-component DFG and inter-procedural CFG to conduct the flow-sensitive and context-sensitive data flow analysis. Such integrated control and data flow analyses approach was proved to be both practical and effective to address security problems. Yang et al. [232] proposed a topic-specific approach to generate sensitive data flow signatures. This method had much less patterns and was much better to characterize malapps than the overall data flow signatures.

11) FILE PROPERTY

File properties refer to features found in apps' important files, such as the '.so' and '.zip' files, the smali files, the suspicious files and so on.

Android apps are distributed as .apk compressed files. Compressed files can reduce the number of download files when installing an app. However, because the compressed files don't contain restrictions of data type, sometimes they are used to carry malicious payloads as .zip files and .so files. As a result, some work utilizes presence or absence of .zip files and .so files as features. For example, Roy et al. [29]

selected the presence of '.so' or '.zip' files as features for Android malapp detection. While Chakradeo et al. [14] detected malapps based on the presence and absence of zip files inside the main app archive. They trained over 15,000 apps from Google Play and 732 known malapps. The experiments showed that their method can find 95% of malapps, and cost 13% of the non-malicious apps on average across multiple markets.

Besides the features mentioned above, Varsha et al. [69] chose the number of smali files as features. Lindorfer et al. [235] collected static features including the presence suspicious files, such as native (shared) libraries, native executables and shell scripts embedded in the apk's resources.

12) SYSTEM COMMAND

System command is a directive of a program to perform a specific task. Since system commands can perform some special task, attackers always help themselves by writing system commands in malapps. For example, when the malapp gets these commands, such as 'chmod', 'chown', 'mount', 'insmod', 'su', and 'bash', it can escalate privilege, root devices or execute malicious shell scripts after obtaining the admin privilege of the mobile devices. Therefore, some work takes system command as a kind of features.

Some researches showed that the system commands can discriminate a malicious app from a benign one. Kate et al. [31] extracted API calls and Android commands from .smali files. They used 190 benign samples and 190 malapps to conduct experiments. The results showed that the accuracy achieved 89% with high efficiency. Kang et al. [230] used malicious commands and other static features to characterize apps' behaviors. Their method finally showed detection performance with accuracy of 98%. Yerima et al. [55] combined advantages of static analysis with the efficient ensemble of machine learning to detect Android malapps. They selected command sets, API calls and permissions as features. They extracted the commands that enabled malapps to escalate privilege, root devices or execute malicious shell scripts at run time and the commands used for stealthily installing additional malicious packages. The experimental results showed that the detection accuracy reached as 97.3%–99% with very low FPR. Yerima et al. [65] chose API calls, Linux system commands and permissions as features. They searched for Java Realtime.exec commands, through which persistent background child processes that contained malicious payload can be launched.

13) NATIVE CODE

Native code refers to the programming code configured to run on a specific processor. Native code used on a processor generally doesn't run on an emulator unless it's allowed. It might be hard to analyze native code, because it is not supported by many analysis tools. Hence, attackers may try to hide parts of their apps' functionality in the native code. Moreover, native code may be used to exploit vulnerabilities of the Android system. Therefore, some work believes that native code can discriminate a malapp from a benign one.

Chakradeo et al. [14] believed that it was likely to be malicious that an app spent lots of resources. They utilized permissions, intents and native code to complete market-scale triage. Lindorfer et al. [235] collected many types of static features including native code via the Java Native Interface and Dalvik bytecode. Hahn et al. [59] added native code as a feature to detect malapps. The experiments results didn't prove the effectiveness of native code on itself. DroidNative [79] was a malapp detection system that operated at the native code level. It can detect malapps embedded in bytecode or native code, and achieved a detection rate of 93.57% with FPR as 2.7%. DroidSieve [233] extracted static features derived from resources including native code. DroidSieve consisted of two parts: malapp detection, achieving up to 99.82% accuracy with zero FPR, and family identification of obfuscated malapps, which achieved 99.26% accuracy.

14) OTHER STRINGS

One of the widely used techniques in classic malapp detection is analyzing strings available in the file. Some work extracts printable strings in Android files, such as menus in the apps. Sayfullina et al. [39] presented a scalable and high accurate method for malapps classification. They extracted features from Android application package (APK) files including strings. They used Normalized Bernoulli (NB) for detection. The method achieved overall accuracy of 91% with 0.1% FPR. Chen et al. [50] extracted nouns of strings, defined in apps' resources files, as keywords. These keywords were more precise than those extracted from the apps' descriptions for malapp detection. Sanz et al. [64] extracted the strings contained in the disassembled code, constructing a bag of words model to generate an anomaly detection model. The system obtained the best accuracy of 83.51%, with an FPR of 27% and a TPR of 94%. Rapoport et al. [211] presented a static analysis tool, Stringoid, which analyzed string concatenations in Android apps to estimate constructed URL strings. They observed that a significant fraction of URLs was only detected by the static analysis. Wang et al. [3] extracted some string features including URLs, IP addresses, file paths and numbers form the disassembled code. They compared the detection results of four kinds of machine learning algorithm, among which the LR classifier yielded the best TPR as 96% with a FPR as 0.06%. In consideration of the requirements for Android malapp detection systems in the real world, Palumbo et al. [234] designed an ensemble approach based on multiple atomic classifiers. The approach extracted lots of string information from Android manifest file, Dalvik executable file (DEX), and resource file. This method was suitable for detecting new, previously unseen malicious Android apps.

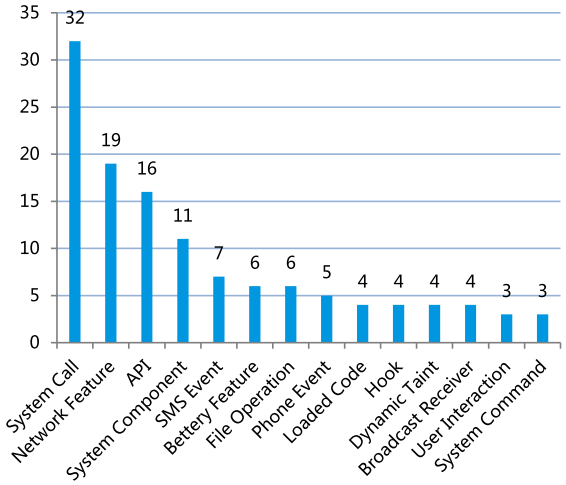


FIGURE 8. The numbers of papers using various of dynamic features.

C. DYNAMIC FEATURE

Dynamic features are the behaviors of the app in interaction with operating system or network connectivity. Dynamic features can be achieved by observing the app's behaviors through its actual execution on real devices. Although the execution may lead to excessive consumption of Android OS, the information observed correctly reflects the app's exact intention. Based on our analysis, out of 234 papers reviewed, 104 papers used dynamic features to conduct their experiments. The numbers of papers using various of dynamic features are shown in Fig. 8. Among the dynamic features, system call is used in 32 papers, more than other dynamic features. The following sections discuss the dynamic features in details.

1) SYSTEM CALL

A user's interaction with Android system through an app generates events in the OS, which are so-called system calls. There are more than 250 system calls in a Linux kernel that are also available in Android. System calls provide useful functions to apps such as network, file, and other related operations. Therefore, analyzing the system calls can obtain accurate information of the apps' behaviors.

There are different approaches for analyzing Android system calls. Studies [127], [160] proposed the suitable system call features used in machine learning for classifying the benign and malicious apps. The system calls generated by each app were captured in a log file using a tool called strace. Burguera et al. [128] considered that monitoring system calls was one of the most accurate methods to understand the behavior of apps, since they provided more detailed low level information. They found that open(), read(), access(), chmod() and chown() were the most used system calls by malapps. With the help of system calls, they obtained the traces of apps' behaviors that can be used to discriminate the malapps from benign ones. The experimental results showed that the system was capable of detecting every malapp execution in self-written malapp, getting a 100% of detection rate for the particular malapp. Wu et al. [193] proposed a method of Android malapp detection using both system call frequency and system call dependency as features. They experimented with several machine learning classifiers: SVM, RF, LASSO and ridge regularization. The results showed that the approach achieved an overall detection accuracy of 93% with 5% benign app classification error. Tong et al. [135] collected known malapps and benign apps to generate patterns of individual system calls and sequential system calls with different calling depth. Then they built up a malicious pattern set and a normal pattern set for malapp detection. The experiment results showed that the integrated detection accuracy rate was above 90%. Jang et al. [143] exploited system calls and system logs as feature vectors. These features performed well in detecting and classifying malware families, reaching 99% accuracy on average. In 2017, Cai et al. [161] firstly presented a systematic dynamic characterization study of Android apps that targeted a broad understanding of the app's behaviors in Android. Moreover, they revealed many meaningful experiments' conclusions for our consideration. Different with existing work, Zhang et al. [164] didn't use all System Calls (SCs) to construct feature vectors for analyzing apps. They first introduced a concept named contribution to quantitatively evaluated SCs relevance for malware identification. They then utilized Markov chains and determinate SCs instead of all SCs to identify malapps. The experimental results demonstrated that their approach possessed the malapp detection ability with high accuracy. Leeds et al. [214] compared static and dynamic analysis of Android apps. They extracted permissions requested at install-time as static features for static analysis. Furthermore, they extracted system calls made at run-time as dynamic features for comparison. Martinelli et al. [168] proposed a method to detect Android malapps. This method was based on convolutional neural network and acquired system calls occurrences through dynamic analysis. They tested the method on a recent dataset composed of 7100 real-world mobile apps, obtaining an accuracy ranging between 0.85 and 0.95. Based on the extracted system calls, Hou et al. [82] constructed the weighted directed graphs and then applied a deep learning framework for newly unknown Android malapp detection. They evaluated the performance of their proposed Deep4MalDroid. They thought that it can be integrated into a commercial Android antimalware software.

As Shown in Fig. 8, system call is the most selected feature among dynamic features. This is because apps cannot directly interact with the Android OS. System calls provide an essential interface between the OS and apps, so they are always used in malapp detection.

2) NETWORK FEATURE

Network operation is another kind of dynamic feature frequently used. Majority of apps need to connect to network to send and receive data, receive updates, etc. Malwares may send users' personal data to attackers through network. Since network operations offer various types of information,

monitoring network operations of apps on mobile devices is an effective way of catching malicious behaviors of apps.

There are different approaches for analyzing Android network features. Amin et al. [130] leveraged two types of dynamic features for malapp detection: system calls and URLs of all remote locations that were connected by apps in specific period. Results showed that the approach based on system calls was able to detect malapps with an accuracy of 87%. Rapoport M et al. [211] firstly collected dynamic data by running 20 randomly selected Android apps, then observed their network activity for the next static analysis of URL strings. Ham et al. [134] collected many types of dynamic features including network features, such as RxBytes, TxBytes, RxPacket, TxPacket. They used SVM and RF for detection and the results of the experiment showed that the system achieved TPR of 94.4% with FPR of 0.4%. AMAL et al. [199] extracted both static features and dynamic features including 65 network features. The network features they chose consisted of 3 groups. Raw network features included counts of unique IP addresses, counts of connections, quartile counts of request size and types of protocols. HTTP features included counts of POST, GET, and HEAD request and the distribution of the size of replied packets. Domain Name Server (DNS) features contained counts of PTR, CNAME, and MX record lookups. Studies [136], [152], [203], [205], [207] extracted network traffic as features. Malik et al. [203] proposed a method that detected malapps on the basis of their DNS queries and the analysis of network traffic logs. Feizollah et al. [152] adopted six kinds of network traffic features, namely frame length, frame number, connection duration, relative duration, source port and destination port. The experimental results showed that using mini batch k-means algorithm performed better than using k-means algorithm for these features in the Android malapp detection.

Despite analyzing network operation is an effective method for Android malapp detection, it has not attracted attention as much as system call features have. Using network operations as feature needs to deal with massive number of network records, which may contain millions of records. Furthermore, analyzing collected network operations requires to deeply understand network architecture.

3) SYSTEM COMPONENTS

Mobile devices have similar components like personal computers, such as CPU, memory and storage. They together make the operating system and mobile device function correctly and efficiently. System components can also provide accurate information about the behaviors of the apps. For instance, since some malapps send users' personal information to attackers, they may take up a lot of CPU.

Some work conducted detection of Android malapp using system components. Bhandari et al. [194] considered memory consumption, CPU consumption and numbers of file operations of each app as features. The proposed method detected 98.4% of the malapps with few false alerts. Canfora et al. [132] selected several types of dynamic features related to the system components: CPU, memory, storage and network features. The system achieved the results of accuracy greater than 99% by using RF classifiers. Zhuo et al. [212] performed dynamic fuzz test for the components and permissions of the app. Their purpose was to judge whether the app existed permission bypass vulnerabilities or denial of service vulnerabilities. Ahmad et al. [241] proposed an approach that automatically targeted triggering the method of interest (MOI). MOI used Inter Component Communications (ICC) for passing data between components.

4) BATTERY FEATURE

This kind of feature gives information regarding to current state of devices' batteries. The battery characteristic can reflect some behaviors of the apps like system components do. Monitoring battery power consumption and battery temperature is the key for malapp detection.

Kurniawan et al. [142] detected malapps through power consumption, battery temperature and network traffic data using three classification algorithms, i.e., SVM, RF and Logistic Model Tree (LMT). The experimental results showed that the best algorithm to work with these three features' combination was RF classifier, achieving 85.6% accuracy. Yang et al. [153] proposed a malapp detection method based on power consumption. They used Gaussian mixture model (GMM) to analyze power consumption. The experimental results showed that the system can achieve accuracy as 79.7%. Michalevsky et al. [155] showed that the information about a user's location can be acquired through reading the phone's aggregate power consumption during minutes. They collected various features including signal strength, voltage, temperature, state of discharge (battery level) and cell identifier. The experiments showed that this approach can reveal much information about the phone's location.

5) PHONE EVENT

Some malapps always send or receive phone calls that threaten users' property security. Therefore, some work takes phone event as another kind of feature. Lindorfer et al. [235] extracted static features and dynamic features including phone events that were represented by the corresponding phone number. Their method correctly classified 98.24% of malapps with less than 0.04% FPR. Ham et al. [134] extracted 32 dynamic features including phone event feature and SMS features. They monitored the apps' behaviors such as sending or receiving phone calls and SMS. The experimental results showed that the system can achieve TPR as 94.4% with FPR of 0.4% based on SVM classifier.

6) SMS EVENT

There exists a kind of malapp that always sends or receives SMS messages. Some SMS messages contain virus links that may harm users' property or information security. Many malapps found in the wild are related to the improper usages

of the SMS functionality. These malapps impose a direct financial cost on the user. Hence, some work take SMS event as a kind of feature. Saracino et al. [184] hijacked the SendTextMessage() and SendDataMessage() methods to extract the information of SMS messages. For example, the phone number corresponding to the sent SMS. During the dynamic analysis, Wang et al. [201] monitored a series of behaviors including SMS and phone events. They extracted 1493 features belonging to these two types. Experiments showed that the anomaly detection with dynamic analysis was capable of detecting zero-day malapps with 1.16% FNR and 1.30% FPR. Wang et al. [207] developed a malapp detection system using SVM classifier based on behavioral features. They extracted 7 types of static features and 12 types of dynamic features including SMS. The experimental results revealed that the overall detection accuracy of the SVC was more than 85% for unspecific mobile malapps.

7) USER INTERACTION

Because users are potential victims of malapps, users' activities form a part of the app's behaviors, such as tapping the screen, long pressing dragging and so on. It's one of the possible solutions for malapp detection to analyze users' interaction with apps.

Some work investigated detection of Android malapps using user interaction. Shabtai et al. [191] extracted both static features and dynamic features including features of user's interaction, such as keyboard/touchscreen pressing and app's start-up. Spreitzenbarth et al. [205] employed the same user interaction features except app's start-up. Their system was able to detect about 94% of the malapps with only 1% FPR. Besides the features mentioned above, Saracino et al. [184] extracted interactive contents on the screen and received inputs from users. The experiments showed that their method effectively detected more than 96.9% of malapps. Alzaylaee et al. [165] implemented a hybrid system by integrating Monkey tool based on random with Droid Bot tool based on state, and it improved code coverage and uncovered more potential malicious behaviors. The results showed that the hybrid approach improved dynamic analysis code coverage and impacted the detection of Android malapps.Some studies employed user interactions for malapp detection. For instance, Martinelli et al. [237] generated a number of user interactions and system events during the app execution.

8) FILE OPERATION

The same as personal computers, mobile devices have file operations such as open, read and write. The operations of some important files can reflect some behaviors of the apps. Therefore, it's an effective way for catching apps' malicious behaviors to monitor apps' file operations.

Some work investigated detection of Android malapps using file operations. Lindorfer et al. [235] extracted file operations as a combined feature. It contained the type (read/write) and the file name. Mohaisen et al. [199] employed many features including 2 kinds of file operation features. The first was the counts for files created, deleted, and modified. The second was the counts for files created in predefined paths like %APPDATA%, %TEMP% and %PROGRAMFILES%. The experiments achieved a precision of 99.5% and recall of 99.6% for certain families' classification. Dash et al. [145] extracted the file access characteristics that contained file name/type, name classes and so on. Moreover, they chose some files as features that may be executed to run or silently install apps. Overall, the proposed system was able to achieve 84% classification accuracy using SVM classifier. Wang et al. [218] proposed a hybrid detection system that was comprised of misuse detection and anomaly detection based on CuckooDroid. They extracted numbers of static and dynamic features including file operation features. The experimental results of their method showed a great accuracy, over 98%.

9) BROADCAST RECEIVERS

Broadcasting is a widely used mechanism for transferring information between apps. Broadcast receiver is a component that filters and receives broadcasts. Because the broadcast receiver can be monitored through dynamic analysis, some work used it as dynamic feature as well. Studies [201], [235] chose dynamically registered broadcast receivers as one kind of features. These broadcast receivers were represented by the intents they registered for. Wang et al. [207] extracted two types of app's components for malapp detection, which were started service and broadcast receiver.

10) LOADING CODE

Some malicious codes are dynamically downloaded from the Internet during execution (i.e., downloading a file from the Internet or calling .elf or .so files). There are two situations: the static payloads include .dex file or .jar files, and the dynamic payloads contain .elf file or .so files. Therefore, some work inspects a given Android app and acquires dynamic loading code when the app is running in a virtual environment or on a real device.

Some work investigated detection of Android malapps using loaded code. Lindorfer et al. [235] extracted loaded code at runtime as feature that was represented by the type of code (either native code or a DEX class). Wang et al. [201] extracted 916 dynamic loaded code as one type of features. The experiments showed that the anomaly detection with dynamic analysis was capable of detecting zero-day malware with 1.16% FNR and 1.30% FPR. Yang et al. [215] proposed a detection method based on Ensemble Learning. They extracted the dynamic loading feature based on static analysis, and then adopted the well-constructed multi-label ensemble learning algorithm to conduct experiments. The experimental results showed that dynamic loading problems can be identified and classified correctly. Moreover, compared to other methods, their detection result was more comprehensive.

11) SYSTEM COMMAND

Commands can be used as another dynamic features. Andro-Dumpsys [236] adopted the usage of system commands of executing forged files for malapp detection. Andro-Dumpsys excluded the system commands with low frequency, and found that some system commands, such as 'chmod', 'insmod' and 'su', were frequently used by malapps. The experimental results demonstrated that Andro-Dumpsys performed well in detecting malapps. It achieved an accuracy of over 99% and classified each malware family with low false positives/negatives.

12) API

APIs can also be used as dynamic features. Rastogi et al. [129] adopted sensitive API monitoring and Kernel-level monitoring in malapp detection. They evaluated 3,968 apps from the official Android Markets and identified exposures of privacy sensitive information of 946 apps. Pirscoveanu et al. [133] applied both API and API sequence as features. They modified the initial sequence of API calls to improve the match between malapps that had similar patterns. They finally got the average accuracy of 98% with the FPR of 4.9%. Ozdemir et al. [198] extracted 4 different feature types from apps. They were complement of each other: Static and Dynamic Native API calls, Static and Dynamic Dalvik Byte API calls. The results showed that using such features increased accuracy and sensitivity of detection operation. Afonso et al. [139] proposed a method of malapp detection based on machine learning and features of API calls and system calls. They evaluated the system with 7,520 apps and obtained a detection rate of 96.66%. Hsiao et al. [141] conducted malapp detection using API call sequence. Somarriba et al. [154] extracted six types of APIs, such as APIs for accessing sensitive data and APIs communicating over the network. The evaluations revealed that the monitoring system did not lose any partial traces, and had a very small impact on the performance of the monitored apps. DroidInjector [169] was a process injection-based dynamic tracking system. It used a process injection technology to attach itself to the target app's process. In a malapp detection system, DroidInjector launched the target app and injected itself to the app. Furthermore, the system will generate a tracking log for the API, and then conduct the behavior-based analysis, getting the results of malapp detection. Alzaylaee et al. [171] extracted many kinds of dynamic features including API calls, presenting an investigation of machine learning based malapp detection. This study performed several experiments to compare emulator based detection with device based detection, the experimental results of device-based analysis obtaining up to 0.926 F-measure with 93.1% TPR and 92% FPR. Chen et al. [172] proposed a framework that used model-based semi-supervised (MBSS) classification scheme built using dynamic Android API call logs. MBSS performed well under the ideal classification setting, with 98% accuracy and very low FPR. Wang et al. [219] showed a novel approach, Droid-AntiRM, to detect possible anti-analysis in Android malapps. Droid-AntiRM used a list of potential-sensitive APIs as target methods and the developers demonstrated it can greatly improve the automated dynamic analysis.

13) HOOK

Hooking obtains control of app execution flow without changing and recompiling the source code. Hook is achieved by stopping the function calls and redirecting them to tailor made codes. By injecting the custom code, any operation can be performed. After that, the main function can be executed and returns the result or it will return to the code that recalls the Hook function. The hooking methods are conducted in two levels, namely 1) Hooking at the user level; 2) Hooking at the kernel level. The majority of written malicious codes for Android OS thus far has targeted the upper layers of Android OS.

Artenstein et al. [156] had a detailed look at Binder, the all-powerful message passing mechanism in Android, and explained how to parse, utilize and exfiltrate the data passed via Binder. Moreover, they demonstrated how the binder functionality could be subverted and integrated into a new kind of Android malapp. Salehi et al. [157] considered Android security from the kernel level and explained the Binder component of Android OS from security point. They designed an active malapp in OS Kernel, and penetrated it into the Binder and controlled data exchange mechanism in Android OS. Therefore, they took control of the whole Android OS through taking control of the Binder approaches.

Tang et al. [167] searched the cache files' security of the high frequency use of Android social apps. They provided a privacy disclosure assessment criterion based on file storage directories and security state machines. They then proposed a protection framework, X-Prcaf (Xposed-based-Protecting-Cache-File), using taint tracking technology, operating system hook technology, and cryptographic technology. Their experiments demonstrated that the X-Prcaf had a good effect on the cache file leaks. Ruan et al. [159] proposed a model, DroidRevealer, based on kernel-level system calls monitoring and ran on real Android devices. DroidRevealer installed a hook in the kernel so as to intercept and interpret system calls and monitored how target data source was used. Moreover, DroidRevealer can reconstruct apps' behaviors in real-time and its experiments proved that their method was acceptable.

14) DYNAMIC TAINT ANALYSIS

Dynamic taint analysis traces data flows from sources to sinks during execution of a program. It allows detection and consequently prevention of flow-based vulnerabilities, such as data leaks or injection attacks. It's a mainstream information control technique.

Shankar et al. [158] developed a framework named Andro-Taint, which applied Dynamic Taint Analysis on Android malapps using automatic tagging and without modification in Android platform. Their Dynamic Taint Analysis algorithm

categorized the apps into risky, benign, malicious or aggressive. AndroTaint covered 90% of malapps and benign in analysis phase with less FP and FN. Schüette et al. [162] proposed a pure application-level dynamic taint analysis. It was not limited to specific platform modifications and can keep up with the precision of platform-level dynamic taint analysis. You et al. [163] considered the characteristic of Android 5.0 and later versions, i.e., Android RunTime (ART). They presented Taint Man, an ART-compatible dynamic taint analysis framework. It can be conveniently deployed on unmodified and non-rooted Android devices. With Taint Man, taint enforcement code was statically instrumented into both the target app and the system class libraries to track data flow and common control flow. Taint Man will be a practical Dynamic Taint Analysis (DTA) framework. Recent years, HTML5 Hybrid apps are becoming more and more popular, meanwhile, more attention must be paid to these apps. Sun et al. [166] proposed a dynamic method to avoid privacy leakage based on dynamic taint tracking in Android.

D. META-DATA

Some work chooses metadata as feature in malapp detection. Metadata is the information that users get before downloading or installing an app, such as the apps' description, apps' rating, developers' information and the information in Android Manifest. Apps' metadata cannot be categorized as static or dynamic features since they have nothing to do with applications themselves.

1) CERTIFICATE FEATURE

Before distributing an app, the developer signs it by his private key. The certificate can be issued by anyone and can be self-signed, but it must be the same for all apps of one author account in the Play Store. Moreover, each app's certificate contains the expedition and expiration date, issuer, subject name, serial number and the country where the certificate is expedited. Therefore, the certificate is useful for identifying malapp author. White and black lists can be created regarding the above information. The certificate has a unique serial number. Based on that, one can check whether certificates are the same or not by comparing the serial number. Some work uses this kind of feature to discriminate a malicious app from a benign one.

Lindorfer et al. [235] extracted the serial number and certificate as features, and these features were used to judged whether the app is self-signed or if its validity period conformed to the release guidelines of the Play Store. Kang et al. [230] adopted serial numbers of certificates in malapp detection to improve the system efficiency. They found that 4% of total certificates collected from malapp were signed as much as 70% of the malapp samples. Jang et al. [236] also extracted serial numbers of certificates as one kind of features. They concluded that the serial number distribution in benign samples was different from that of malapp samples. Lin et al. [238] extracted signature information from the META-INF files, to judge whether the app was from a malicious developer and a malapp family. And it demonstrated that the signature was helpful for malapp detection.

2) DEVELOPER-RELATED AND PUBLISHER-RELATED FEATURES

This type of feature includes some information of developer and publisher, such as developer ID, publisher ID, their emails and webpages. This information allows to create white and blacklists regarding developers' reputation. If an app's developer is recurrent in developing malapp, the app is likely to be malapp. Muñoz et al. [185] captured around 7,569 different developer names in their dataset. They chose these as one type of feature in malapp detection. Some work uses publisher IDs to build advertisement (ad) libraries. This is because publisher IDs can be used to identify whom to pay the ad view revenue. Lindorfer et al. [235] adopted many metadata features including publisher IDs used to identify the publishers that pay the ad revenue. Wang et al. [3] extracted developer information from the certificate, such as the country, email address, organization and so on, and they were used for malapp detection. Martinelli et al. [237] extracted many kinds of static and dynamic features as well as meta-data features including developer reputation for detection. Their framework tested on a set of 10,000 genuine apps, achieving a detection accuracy of 99.7% with negligible FPR.

3) INTRINSIC APPLICATION FEATURES

This type of feature involves app's title, size (in bytes), code version, number of images and files, and the date of creation, upload and update at Google Play. Feldman et al. [231] extracted the low version numbers of apps because they observed that malicious apps tended to have lower version numbers compared to benign apps. They found that 247 malicious apps had a 1.x app version number, while only 84 benign apps exhibited this characteristic.

4) SOCIAL-RELATED FEATURES

This type of feature includes relevant feedback that is collected from users and available at the market. Features like number of total votes and apps' rating make up this feature set. Saracino et al. [184] extracted some reputation metadata as features, such as user scores, marketplace and download number. These features are used for performing a preliminary risk assessment of the app.

5) JAVA PACKAGE NAME

The Java package name can uniquely identify the apps in the Google Play Store and many alternative markets. White and black lists may also be created regarding this feature. Lindorfer et al. [235] extracted the Java package name that uniquely identifies apps in app markets.

6) HYBRID META-DATA INFORMATION

Different from previous work on metadata, Wang et al. [186] proposed a lightweight method based on apps' metadata for

malapp detection in Android. Mmda statically analyzed the app's executable file and took permissions, hardware features and receiver actions as app's metadata. In evaluation, Mmda with RF classifier was up to 94% malapp detection rate. Compared with popular anti-virus scanners VirusTotal, Mmda with RF had a better detection rate among the most recent dataset. Martín et al. [187] proposed a fast and accurate method ADROIT employing meta-information available on the app store website and in the Android Manifest. They combined and applied text mining and machine learning approach to detect malapps. ADROIT achieved 94% accuracy rate. Akhuseyinoglu et al. [188] utilized free public metadata provided by Google's official application market as the feature set for training supervised classification algorithms, Navie Bayes and the final experimental results showed that adding the permissions in the official market metadata as a predictor improved the accuracy of the test. Moreover, they developed an app AntiWare to demonstrate the effectiveness of the proposed method. Wu et al. [189] presented a system PACS (Permission Abuse Checking System), that detected the permission abuse problem for the apps. PACS firstly used machine learning and data mining techniques to classify the apps into different categories by considering apps' metadata information, e.g., the reviews, descriptions, etc. Then, it used Apriori algorithm to mine frequent patterns of the app within the same category, getting the maximum frequent item sets and constructed the permission feature database. Finally, PACS was used to detect the unknown apps of the abuse of authority, and the experiment results achieved 77.6% accuracy rate.

7) TOPIC FEATURE

The topic of a Android application, such as Body building, Sport live, Puzzle game and so on, denotes its classification in the app markets. These information can be obtained from the app's description. Yang et al. [232] used an advanced topic model, adaptive Latent Dirichlet Allocation (LDA) with Genetic Algorithm (GA), to cluster apps according to their descriptions. They conducted an empirical study on 3691 benign and 1612 malicious apps, and grouped them into 118 topics.

V. FEATURE SELECTION METHODS

Feature selection approach, also called variable selection approach or attribute selection approach, can be used to choose features that are most relevant to the predictive modeling problems [242]–[244]. Some irrelevant features and redundant features may appear in feature sets. Irrelevant features should be reduced because they will have low correlation with the class. Redundant features should be screened out as they will be highly correlated with one or more of the remaining features. Since feature selection approach can remove irrelevant and redundant features, it usually gives a good or better accuracy whilst requiring less data.

There have been a lot of features in malapp detection. Some work believes that choosing adequate features is an important step in malapp detection because appropriate features determine effectiveness and results of an experiment. Out of 234 papers reviewed, 31 papers used feature selection method. We summarize the following five approaches for feature selection based on the reviewed papers.

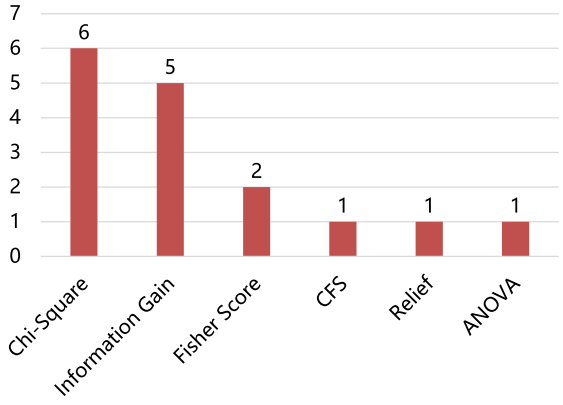


FIGURE 9. The number of the usage of filter approaches.

A. SELECTION BASED ON FILTER APPROACH

In the filter approach, features are selected by performing some statistical analysis on features and labels. Filter approaches select features without generating learning models. They are based only on general features like the correlation with the variable to predict. Filter methods suppress the least interesting variables and the other variables will be used to classify or to predict data. These methods are particularly effective in computation time and robust to overfitting. There are several filter approaches always used for malapp detection. The number of the usage of filter approaches is shown in Fig. 9.

Chi-Square is a popular feature selection method. In statistics, the Chi-Square test is applied to test the independence of two events. Events A and B are defined to be independent if P(AB) = P(A)P(B) or, equivalently, $P(A|B) = P(A)$ and $P(B|A) = P(B)$. From the definition of Chi-Square, the use of chi-square technique in feature selection can be easily inferred. Suppose there is a target variable (i.e., the class label) and some features of the app. Now, we calculate chi-square statistics between every feature and the target variable, and observe the existence of a relationship between the variables and the target. If the target variable is independent of the feature, we can give up that feature. Otherwise, the feature is very important. Shabtai et al. [191] extracted many types of features. Three feature selection methods were applied in Study [191]: Chi-Square, Fisher Score and Information Gain. They found that Chi-Square and Information Gain graded the same top 10 selected features with a very similar rank. When the authors used top 10 features that are selected through Chi-Square, they achieved the accuracy of 96.3% that was higher than those of using more features. Sheen et al. [41] adopted Chi-Square algorithm to select features. Based on the top 20 features, the system finally achieved 93.21% accuracy.

Information Gain (IG) measures the amount of information obtained for class prediction by observing the presence or absence of features. Therefore, in malapp detection, IG measures the amount of information that a feature brings to detection system. The more the amount of information the feature carries, the more important the feature is. Shabtai et al. [191] applied three feature selection methods: Chi-Square, Fisher Score and IG. When they used top 10 features selected through IG, the method achieved the accuracy of 96.8% that was higher than those of using the features selected through Chi-Square and Fisher Score. Sheen et al. [41] adopted IG algorithm to select features. Based on the top 20 features, the system finally achieved 93.96% accuracy.

Fisher score analysis is a simple and effective technique to select the most relevant feature from the dataset. Its main purpose is to determine a subset of features. In the space where the feature data spans, the distances between data points in different classes are as large as possible, while the distances between data points in the same set are as small as possible. Because Fisher Score evaluates features individually, it cannot handle feature redundancy. Shabtai et al. [191] used top 10 features selected through Fisher score, and the accuracy was up to 96.2% that was higher than those using more features. Lindorfer et al. [235] used the Fisher score to reduce the dimensionality of feature vector and used the most useful features. The experiments showed that the system achieved highest accuracy with a set of the 27,808 highest ranked features, 18,335 of which were dynamic and 9,473 of which were static.

The RELIEF algorithm calculates the quality of features according to how well their values distinguish between apps that are near to each other. For this purpose, given a randomly selected app, $x_i = \{x_{1i}, x_{2i}, \ldots, x_{ni}\}$, RELIEF searches for its two nearest neighbors: Nearest-hit H found from the same class, and nearest-miss M found from a different class. After that, it updates the quality estimate for all the features according to the values of x_i, M, and H. The RELIEF is robust to feature interactions and can be applied to binary or continuous data. Sheen et al. [41] employed three feature selection methods: Chi-Square, Relief and Information Gain. The experimental results showed that the top 20 features adopted from RELIEF got the best accuracy, up to 94.97%.

Correlation based Feature Selection (CFS) is a simple filter algorithm that ranks features according to a correlation based on the heuristic evaluation function. Therefore, CFS consists of a search algorithm and a function that evaluates the merit of features. Since good feature subsets contain features highly correlated with the class and they are uncorrelated with each other, CFS measures the usefulness of individual features for predicting the class label along with the level of inter-correlation among them. Xu et al. [10] firstly extracted 121,621 ICC-related features in total. They applied CFS to remove the redundant features and finally chose 5,000 ICC-related features that were used for the classification of malicious and benign apps. It achieved accuracy of 97.4% with 0.67% FPR.

One-way analysis of variance (ANOVA) is a novel feature selection method. ANOVA is a technique used to compare means of three or more samples. In the ANOVA table, entries SSE (Sum of squared errors), SST (Treatment sum of squares) and Total SS (Total Sum of Squares) can be found. Proportion of variance explained by the feature can be calculated as follows:

$$Variance = \frac{SST}{totalSS}$$

The higher ratio is, the more proportion of variance the feature can explain in the data. It follows that the features with high proportion should be selected. Qiao et al. [63] compared two feature selection methods: (ANOVA) and Support Vector Machine-Recursive Feature Elimination (SVM-RFE) which is a wrapper method. When using the ANOVA, the system achieved 91.36% accuracy.

B. SELECTION BASED ON WRAPPER APPROACH

The wrapper approach offers a simple and powerful way to address the problem of feature selection. Wrapper approach considers the feature selection as a search problem, where different combinations are prepared. Since a combination should be compared to other combinations, predictive models are formed to evaluate combinations of features. Then it assigns a score to each combination based on model accuracy. Although the wrapper approach may obtain better performances, there are some disadvantages of wrapper approach. It will require greater computational resources as well as more computation time when the number of features is large. Meanwhile, the overfitting risk will increase when the number of observations is insufficient.

One wrapper approach, Support Vector Machine-Recursive Feature Elimination (SVM-RFE) was used in study [63] for feature selection. SVM-RFE is a powerful feature selection algorithm. SVM-RFE computes the ranking weights for all features and gives a list of features in the order of weights. The algorithm then will remove the feature with smallest ranking weight, while retaining the features of important impact. It is a good choice to avoid overfitting when the number of features is high. However, it may be biased when there are highly correlated features. Qiao et al. [63] compared two feature selection methods: ANOVA and SVM-RFE. When using the SVM-RFE, the system achieved 91.52% accuracy, a little higher than that of using ANOVA.

C. SELECTION BASED ON THE USAGE OF FEATURE

Some work believes that the features that used more in malicious apps will be more effective than the features that used more in benign apps. Aafer et al. [12] firstly extracted 8,375 APIs. Then they took only the APIs whose usage in the malapp set was higher than in the benign set and finally reduced the features to 169 APIs. They achieved the highest accuracy of nearly 99% using KNN. Chen et al. [196]

compared the permissions requested by the malicious apps with the permissions requested by benign ones and ultimately selected 59 out of 120 permissions as features. Chuang et al. [56] ranked the APIs according to their usage difference in percentage. If an API is used by $x\%$ of malicious apps and $y\%$ of benign apps, the API will be ranked according to $(x - y)\%$. Then they select the top k APIs according to the rank such that the number k satisfied a threshold of 10% difference in API usage frequency. The experiment showed that the system achieved 96.69% accuracy with 2.5% false positive rate in detecting unknown malapps. Hang et al. [71] weed out the features that most normal apps had to get effective features for malapp detection. Varsha et al. [69] employed a feature selection method called entropy based Category Coverage Difference (ECCD). In terms of this method, if the feature appears only in one class, then the entropy will be minimum (equals 0). It indicates that this feature may be a good discriminant feature. If the feature appears in all class with same frequency, then the entropy will be maximum. Varsha et al. [69] got accuracy of 98.14% at a feature length of 234 using RF classifier.

D. SELECTION BASED ON LOGISTIC REGRESSION

Logistic regression is a linear classifier whose parameters are weights, usually in terms of weight vector ω, and the regularization parameter. After training logistic regression, ω is estimated, and the value of each weight represents how important that weight is for classification.

Wu et al. [49] pre-generated an API list ranked with malicious weight values using the LR algorithm. For the classifier evaluation, they selected top k (50, 100, 150, 200, 250, and 300) dataflow-related APIs as feature vectors and generated the classification model. The experiment result showed that the contribution of the top 150 APIs for identifying malapp was the highest. Muñoz et al. [185] used Logistic Regression model along with the Step Akaike Information Criterion for feature selection. The feature selection algorithm reduced the number of features to the eight most relevant ones. The experiments finally achieved the 97% accuracy.

E. SELECTION BASED ON OTHER APPROACHES

Since feature matrices generated by the system call dependency exhibited high sparsity, Dimjavsevic et al. [131] removed all columns without a nonzero element from the matrices. The experiment showed that this method obtained detection accuracy of 93% with FPR of 5%.

Zhao et al. [38] took the feature dataset as input, and ran FrequenSel to decide whether a feature satisfied with the specific conditions or not. These conditions required that the frequency and coverage of a feature should exceed several predefined thresholds. The typical features they collected were not only frequently used by malapp, but also had a certain coverage in the feature dataset. They only used 398 features in the experiments, which was much less than over 32,000 features they extracted. Thier method finally got 97.8% accuracy which is better than most of the scanners.

Mohaisen et al. [199] ran the recursive feature elimination (RFE) algorithm which ranks all features from the most important to the least important feature. First, the estimator was trained on the initial set of features and weights were assigned to each one of them. Then, features whose absolute weights were the smallest were pruned from the current set features. The procedure was recursively repeated on the pruned set until the desired number of features to select was eventually reached. Mohaisen et al. finally chose the top 65 features for malapp detection.

A specific method called ClassifierSubsetEval from Weka machine learning tool was used to feature selection. Narudin et al. [136] selected 6 features out of 11 after applying feature selection algorithm.

Wang et al. [201] used VarianceThreshold for feature selection in anomaly detection. VarianceThreshold is a simple baseline approach to feature selection. It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e. features that have the same value in all samples.

Yerima et al. [65] utilized Mutual Information (MI) calculation to rank the extracted features. After calculating the MI for each feature, they ranked features from largest to smallest in order to select those maximized MIs to get optimal classifier performance. When they chose the top 10 features for experiments, the system achieved the accuracy of 97.43%.

Liu et al. [67] applied an integrated feature selection approach based on the principle of self-learning to extract important features. The integrated feature selection method includes two main steps: generation of training subsets to establish balanced training subsets and merging feature subsets to obtain a global feature set. The method can decrease the dimension of features and reduce the risk of information losing.

Apart from the feature selection methods, there also exist many methods for the classification or detection of anomalies, intrusions or malapps [245]–[253].

VI. RELATED SURVEYS

There have been thousands of features used for Android malapp detection. However, to the best of our knowledge, there is no systematic literature review on the proposed features yet. Several related surveys have been conducted. But they don't carry out comprehensive analysis of the existing features.

Sadeghi et al. [7] provided a survey of the existing approaches for Android security analysis. They analyzed the results of 336 research papers published in diverse journals and conferences. They constructed a taxonomy by performing a ''survey of surveys'' and conducting an iterative content analysis over papers they collected. Then they applied the taxonomy to classify the papers about Android security. They finally conducted a comprehensive analysis on different concepts in the taxonomy, and presented current trends

and gaps in the existing papers. Simultaneously, they underlined key challenges and opportunities that will give a direction of future research efforts. Faruki et al. [254] discussed the Android security enforcement mechanisms and threats to the existing security enforcements and related issues. In addition to the existing detection methods, this paper referred the malapp growth timeline between 2010 and 2014, and stealth techniques employed by the malapp developers. Cooper et al. [255] gave an overview of malapp characteristics commonly found in the market. The malapps were viewed in terms of characteristics, such as the requested permissions, API call sequences and nature of arguments. Then, they showed the mitigation approaches that are currently presented in the papers and highlighted the advantages and disadvantages. Rashidi et al. [256] discussed the existing Android security threats and existing security enforcement solutions. The papers they selected were primarily behavior-based and focused on tracing the apps' system calls and analyzing the activities. Li et al. [8] gave a literature review that analyzed around 90 papers about software engineering, programming languages and security venues. They presented a view of papers that statically analyzed Android apps, and highlighted the trends of static analysis approaches. They pointed out where the work should focus on and gave a direction that future researches still need to do.

Although the above previous work involves the analysis of Android apps, they focus on either macroscopic security analysis of Android app or comparison of existing tools. In this paper, we aim to investigate security of Android apps in detail. We focus on the features and the feature selection methods used in existing work for malapp detection. We present a taxonomy of existing features and feature selection methods. In addition, we provide directions for future research in next section.

VII. DISCUSSIONS AND DIRECTIONS FOR FUTURE WORK

In the previous sections, we look back at the related work with respect to features used for Android malapp detection. Existing features can be classified into 3 categories, i.e., static features, dynamic feature and meta-data based features, each of which contains many subsets. Based on our survey, it is easy to find out which features are frequently used. In this section, we first give some discussions of surveyed papers, and then provide directions in the field of Android malapp detection.

A. DISCUSSIONS

The analysis of Android apps has gained much attention. From our survey, we can see that it has become and remained a hot topic since 2014. We have following findings from our survey.

1) THE TRENDS OF THE ANALYSIS OF ANDROID APPS

Android malapp detection can be implemented through static analysis, dynamic analysis, or hybrid analysis. Fig. 10 depicts the trends of the analysis of Android apps from 2011 to 2018. It is obvious that static analysis technology dominates

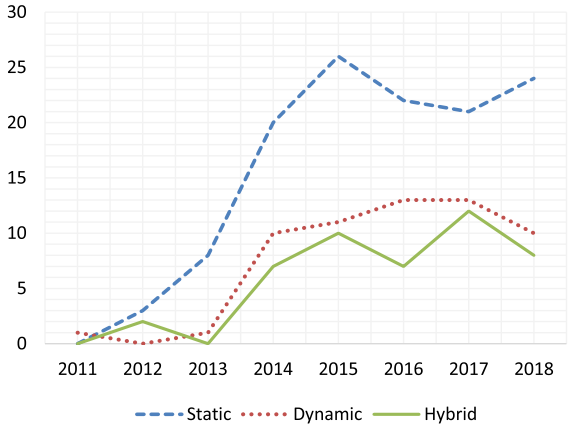


FIGURE 10. The Trends of The Analysis of Android Apps.

the field of Android malapp detection. Meanwhile, related work has paid an increasing attention on dynamic and hybrid analysis technologies in recent years. Dynamic analysis well makes up for some deficiencies of static analysis, presenting a slowly rising trend. Hybrid analysis combines both merits of static and dynamic analysis.

2) PERFORMANCE OF EXISTING FEATURES

Most existing work extracts multiple features rather than a single feature for accurately detecting Android malapps. In our previous work [3], we have made a comparison between 11 single-type feature sets and 3 combined feature sets, showing that the latter generally outperforms the former. As shown in table 3, we exhibit the accuracy of detection based on some features that has been shown in state-of-the-art work. Although the detection results depend on the dataset used, these results reflect the efficiency of their selected features. A high detection rate can be obtained by extracting some features, such as API, Permission, Filtered Intent, System Call and so on. These effective features are recommended for detecting Android malapps. Some other features like 'Battery Feature' and 'Other Strings' is vulnerable to detect malapps, but they may be helpful to inspect newly unknown apps.

B. DIRECTIONS

As mentioned in Section I, there are many key issues that future work needs to resolve. For these issues, we provide some directions in the field of Android malapp detection.

1) The number of extracted features is generally too large to deal with. Well-discriminated features characterize apps' behaviors more accurately, thus detecting and categorizing based on these features can be more effective and efficient. The future work should hence focus on exploring and refining more effective and representative features from apps. In addition, future work can remove redundant features and filter out more irrelative features through optimizing the feature selection approaches.

TABLE 3. The accuracy of detection based on individual feature.

Taxonomy	Feature	Accuracy	Paper
Static Features	N-C, API, Per, etc.	99.82%	Suarez et al. [236]
	N-C, CFG	93.57%	Alam et al. [79]
	Intent, Per, MD5	99.00%	Song et al. [43]
	Intent, App Comp	97.40%	Xu et al. [10]
	Intent, Per	95.50%	Feizollah et al. [76]
	API, CFG	98.76%	Dam et al. [84]
	API, N-A, Per	98.00%	Zhao et al. [38]
	API	95.94%	Hu et al. [24]
	APP Comp, Intent, Per, etc.	98.14%	Varsha et al. [69]
	APP Comp, Per	97.00%	Mohsen et al. [92]
	APP Comp, Per	90.00%	Feldman et al. [234]
	Opcode, Per	98.00%	Ali et al. [85]
	S-C, API	97.30%	Yerima et al. [55]
	F-P, Per, N-C, etc.	95.00%	Chakradeo et al. [14]
	F-P, API, etc.	94.62%	Apvrille et al. [9]
	CFG	94.00%	Allix et al. [28]
	Other Strings	83.51%	Sanz et al. [64]
Dynamic Features	Sys-Component	99.00%	Canfora et al. [132]
	Sys-Call, API, Per	99.00%	Jang et al. [143]
	Sys-Call, UI	95.00%	Fabio et al. [168]
	API	98.00%	Pirscoveanu et al. [133]
		98.00%	Chen et al. [172]
	API, Sys-Call	96.66%	Afonso et al. [139]
	UI, Sys-Call, etc.	96.90%	Saracino et al. [240]
	Network Traffic	99.90%	Chen et al. [177]
	Dynamic Taint	90.00%	Shankar et al. [158]
	Battery Feature	85.60%	Kurniawan et al. [142]
		79.70%	Yang et al. [153]
	File Operation, etc.	84.00%	Dash et al. [145]
Remarks	N-C: Native Code; Per: Permission; Intent: Filtered Intent; App Comp: App Component; CFG: Control Flow Graph; N-A: Network Address; S-C: System Command; F-P: File Property; UI: User Interaction		

2) Collecting more representative malapps and analyzing their characteristics is very important for their detection. These behaviors may be helpful to amend the distinguished feature sets.
3) To deal with the issue of large size of APKs, it is imperative to explore more effective features that are easily to extract in future work, so that the difficulty of extracting features will not be subject to APKs' size.
4) The number of apps continues to grow rapidly. In the future, malapp detection can combine with AI or machine learning based algorithms, such as deep learning, to make the detection more intelligent, so as to facilitate automated detection and to ease the management of app markets.
5) In terms of static features, it may need to pay more attentions to the features that perform well in persistence, such as our proposed platform-define features. It is also feasible to extract some underlying features, such as kernel-level features, so that malapp detection has relatively low dependence on platforms and apps.
6) In view of the obfuscated Android apps, it's necessary to extract some obfuscation-invariant features. An investigation [257] indicates that there are several common obfuscation techniques used by Android apps, namely, identifier renaming, string encryption, Java reflection and packing. Besides, control flow obfuscation technology is also widely used in Android apps. There has been some detection methods for detecting obfuscated apps with different obfuscation techniques. The same as Android malapps detection, obfuscation detection can also extract relevant features from APK files and employ machine learning algorithms for classification [258]. The features should be diverse for different obfuscation techniques [233]. For example, with identifier renaming, the apps' identifiers would be replaced by some single characters (like 'a' , 'b') or strings with repetitive characters, e.g. 'aaaa', 'bbbb'. Therefore, the identifiers can be extracted for detecting identifier, such as class names, methods names, fields names and so on [259]. To detect string encryption, some features can be captured by analyzing constant strings of an app before and after string encryption. As for control flow obfuscation detection, we can extract features from the control flow graphs (CFG) of apps.
Through distinguishing the obfuscation techniques used by obfuscated apps, distinctive and targeted features can be extracted for detecting obfuscated malapps, so as to improve the detection efficiency. For instance, it is necessary to dynamically analyze apps for detecting malapps that obfuscated by control flow or Java reflection technologies, while it is only needed to extract some structural features using static analysis method, e.g. function call graph, to detect malapps with identifier renaming. Hence, effective detection approaches should be proposed corresponding to different obfuscation techniques, in order to realize automatic detection on obfuscated malicious apps.
7) For dynamic analysis, it requires to automatically traverse all possible execution paths so as to extract as many as effective features for the detection of malapps. It is imperative to improve or develop a more sonicated tool for automated dynamic analysis.
8) For the problem of sparse matrix, it requires to develop approaches to processing high dimensional sparse vectors effectively and efficiently.

In addition, we believe that there should be a publicly available test dataset so that researchers can verify their proposed methods, from the features to the classification algorithms. The vetting for malapp still needs further exploration and research.

VIII. CONCLUDING REMARKS

In this paper, we follow the systematic literature review process, and conduct a systematic survey of the features used in state-of-the-art research literatures aiming at Android malapp detection. First, we review thousands of papers published from January 2011 to February 2019, and finally select 236 papers from them. Second, we present a taxonomy of

the features that the related employs. Third, based on the survey, we highlight the issues of constructing features for malapp detection. Finally, we summarize the trends in the field of Android malapp detection and provide some directions for future work.

This survey shows that the existing work should explore and refine well-discriminated features from numerous static features, and pay more attention to dynamic analysis and hybrid analysis to make up for inherent defects in static analysis. We find that existing work easily overlooks some features, such as native code, dynamical loading code and dynamic loading library and these can be amended to form a more effective feature set for the detection of malapps.

From our investigation, we find that (1) The analysis of Android apps has gained much attention. It has become and remained a hot topic since 2014; (2) As code protection techniques (e.g., code obfuscation or encrypted shell protection) have been widely used and the size of apps has become increasingly large, effective features are not easy to extract with traditional static analysis methods. It calls for subtle techniques like de-obfuscation or anti-packaging so as to construct more effective features from apps; (3) As functions of apps have become increasingly powerful and the behaviors of malapps have become increasingly sophisticated, dynamic analysis is more effective than static analysis in terms of feature extraction for the detection of malapp. However, extracting features by dynamic analysis is too time-consuming to vet a very large number of apps. In addition, traversing all possible execution paths is a challenging task. It is thus imperative to develop more effective tools to simulate triggering all possible events in apps so as to construct representative features; (4) The meta-data features do not show a good ability for discriminating malicious apps from benign ones. However, this kind of features can be for rapid initial screening and analysis of apps, so that the detection can be accelerated; (5) The number of features extracted from an app may be up to a million, it is thus imperative to perform feature selection in order to remove redundant or irrelevant features, so that massive apps in the market can be efficiently processed; (6) As the number of features as well as the number of apps increases dramatically, effective artificial intelligence or machine learning approaches are required to process very high dimensional vectors representing apps. Moreover, the detection of Android apps can be performed with deep learning models to make detection more intelligent and automated, especially for the detection of unknown malapps; (7) Only a small portion of state-of-the-art work have made their features publicly available. Publicly sharing specific selected features is impelled so that the detection methods can be fairly evaluated and compared.

REFERENCES

[1] *360 Internet Security Center*. Accessed: Feb. 18, 2019. [Online]. Available: http://zt.360.cn/1101061855.php?dtid=1101061451&did=610100815

[2] *Data of Appbrain*. Accessed: Feb. 20, 2019. [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps

[3] X. Wang, W. Wang, Y. He, J. Liu, Z. Han, and X. Zhang, "Characterizing Android apps' behavior for effective detection of malapps at large scale," *Future Gener. Comput. Syst.*, vol. 75, pp. 30–45, 2017.

[4] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in Google play," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 222–235. [Online]. Available: https://doi.org/10.1145/3274694.3274726

[5] X. Zeng, G. Xu, X. Zheng, Y. Xiang, and W. Zhou, "E-AUA: An efficient anonymous user authentication protocol for mobile IoT," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 1506–1519, Apr. 2019.

[6] G. Xu, Y. Zhang, A. K. Sangaiah, X. Li, A. Castiglione, and X. Zheng, "CSP-E2: An abuse-free contract signing protocol with low-storage TTP for energy-efficient electronic transaction ecosystems," *Inf. Sci.*, vol. 476, pp. 505–515, Feb. 2019. doi: 10.1016/j.ins.2018.05.022.

[7] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software," *IEEE Trans. Softw. Eng.*, vol. 43, no. 6, pp. 492–530, Jun. 2017.

[8] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of Android apps: A systematic literature review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, Aug. 2017.

[9] L. Apvrille and A. Apvrille, "Identifying unknown Android malware with feature extractions and classification techniques," in *Proc. Trustcom/Bigdatase/Ispa*, Aug. 2015, pp. 182–189.

[10] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on Android," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 6, pp. 1252–1264, Jun. 2016.

[11] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in Android applications for malicious application detection," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 11, pp. 1869–1882, Nov. 2014.

[12] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Sydney, NSW, Australia: Springer, 2013, pp. 86–103.

[13] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.

[14] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2013, pp. 13–24.

[15] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE/ACM IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 426–436.

[16] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 576–587.

[17] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, Nov. 2014, pp. 1105–1116. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660359

[18] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2013, pp. 185–196.

[19] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New Orleans, LA, USA, Dec. 2014, pp. 56–65. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664275

[20] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 175–186.

[21] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. IEEE/ACM IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 280–291.

[22] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proc. ACM Symp. Access Control Models Technol.*, 2012, pp. 13–22.

[23] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Scottsdale, AZ, USA, Nov. 2014, pp. 1329–1341. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660357

[24] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, "MIGDroid: Detecting app-repackaging Android malware via method invocation graph," in *Proc. Int. Conf. Comput. Commun. Netw.*, 2014, pp. 1–7.

[25] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Comput. Secur.*, vol. 49, pp. 255–273, Mar. 2015.

[26] T. Shen, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang, "Detect Android malware variants using component based topology graph," in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Sep. 2014, pp. 406–413.

[27] X. Liu and J. Liu, "A two-layered permission-based Android malware detection scheme," in *Proc. IEEE Int. Conf. Mobile Cloud Comput., Services, Eng.*, Apr. 2014, pp. 142–148.

[28] K. Allix, J. Klein, R. State, and Y. L. Traon, "Large-scale machine learning-based malware detection: confronting the '10-fold cross validation' scheme with reality," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2014, pp. 163–166.

[29] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, "Experimental study with real-world data for Android app security analysis using machine learning," in *Proc. 31st Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2015, pp. 81–90. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818038

[30] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust statistical feature signature for Android malware detection," in *Proc. Int. Conf. Secur. Inf. Netw.*, 2013, pp. 152–159.

[31] P. M. Kate and S. V. Dhavale, "Two phase static analysis technique for Android malware detection," in *Proc. 3rd Int. Symp. Women Comput. Inform. (WCI)*, Kochi, India, Aug. 2015, pp. 650–655. [Online]. Available: http://doi.acm.org/10.1145/2791405.2791558

[32] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 45–54.

[33] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[34] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. Inf. Secur.*, Aug. 2012, pp. 62–69.

[35] H. V. Nath and B. M. Mehtre, "Static malware analysis using machine learning methods," in *Proc. Int. Conf. Secur. Comput. Netw. Distrib. Syst.* Thiruvananthapuram, India: Springer, 2014, pp. 440–450.

[36] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 281–294.

[37] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and api calls," in *Proc. IEEE Int. Conf. Tools Artif. Intell.*, Nov. 2014, pp. 300–305.

[38] K. Zhao, D. Zhang, X. Su, and W. Li, "Fest: A feature extraction and selection tool for Android malware detection," in *Proc. Comput. Commun.*, Jul. 2016, pp. 714–720.

[39] L. Sayfullina, E. Eirola, D. Komashinsky, P. Palumbo, Y. Miche, A. Lendasse, and J. Karhunen, "Efficient detection of zero-day Android malware using normalized Bernoulli naive Bayes," in *Proc. IEEE Trustcom/Bigdatase/Ispa*, Aug. 2015, pp. 198–205.

[40] G. Canfora, A. de Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of opcode ngrams for detection of multi family Android malware," in *Proc. Int. Conf. Availab., Rel. Secur.*, Aug. 2015, pp. 333–340.

[41] S. Sheen, R. Anitha, and V. Natarajan, "Android based malware detection using a multifeature collaborative decision fusion approach," *Neurocomputing*, vol. 151, pp. 905–912, Mar. 2015.

[42] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. A. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *Proc. 19th Eur. Symp. Res. Comput. Secur. (ESORICS)*, Wroclaw, Poland, Sep. 2014, pp. 163–182. doi: 10.1007/978-3-319-11203-9_10.

[43] J. Song, C. Han, K. Wang, J. Zhao, R. Ranjan, and L. Wang, "An integrated static detection and analysis framework for Android," *Pervasive Mobile Comput.*, vol. 32, pp. 15–25, Oct. 2016.

[44] K. Sokolova, C. Perez, and M. Lemercier, "Android application classification and anomaly detection with graph-based permission patterns," *Decis. Support Syst.*, vol. 93, pp. 62–76, Jan. 2016.

[45] L. Cen, C. S. Gates, L. Si, and N. Li, "A probabilistic discriminative model for Android malware detection with decompiled source code," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 4, pp. 400–412, Jul. 2015.

[46] D. Su, W. Wang, X. Wang, and J. Liu, "Anomadroid: Profiling Android applications' behaviors for identifying unknown malapps," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, Aug. 2016, pp. 691–698.

[47] J. Zhu, Z. Wu, Z. Guan, and Z. Chen, "API sequences based malware detection for Android," in *Proc. IEEE Int. Conf. Auton. Trusted Comput. IEEE Int. Conf. Scalable Comput. Commun. ITS Associated Workshops Ubiquitous Intell. Comput.*, 2016, pp. 673–676.

[48] Z. Wang, C. Li, Y. Guan, and Y. Xue, "DroidChain: A novel malware detection method for Android based on behavior chain," in *Proc. Commun. Netw. Secur.*, Sep. 2015, pp. 727–728.

[49] S. Wu, P. Wang, X. Li, and Y. Zhang, "Effective detection of Android malware based on the usage of data flow APIs and machine learning," *Inf. Softw. Technol.*, vol. 75, pp. 17–25, Jul. 2016.

[50] W. Chen, D. Aspinall, A. D. Gordon, C. Sutton, and I. Muttik, "More semantics more robust: Improving Android malware classifiers," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2016, pp. 147–158.

[51] A. Martín, H. D. Menéndez, and D. Camacho, "MOCDroid: Multi-objective evolutionary classifier for Android malware detection," *Soft Comput.*, vol. 21, no. 24, pp. 7405–7415, 2016.

[52] A. Martín, H. D. Menéndez, and D. Camacho, "String-based malware detection for Android environments," in *Proc. Int. Symp. Intell. Distrib. Comput.* Paris, France: Springer, 2016, pp. 99–108.

[53] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of Android malware for effective malware comprehension, detection, and classification," in *Proc. 25th Int. Symp. Softw. Test. Anal. (ISSTA)*, Saarbrücken, Germany, Jul. 2016, pp. 306–317. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931043

[54] Z. Zhu and T. Dumitras, "FeatureSmith: Automatically engineering features for malware detection by mining the security literature," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 767–778.

[55] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy Android malware detection using ensemble learning," *IET Inf. Secur.*, vol. 9, no. 6, pp. 313–320, 2015.

[56] H.-Y. Chuang and S.-D. Wang, "Machine learning based hybrid behavior models for Android malware analysis," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 201–206.

[57] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Comput. Secur.*, vol. 43, no. 6, pp. 1–18, 2014.

[58] L. Li, K. Allix, D. Li, A. Bartel, T. F. Bissyandé, and J. Klein, "Potential component leaks in Android apps: An investigation into a new feature set for malware detection," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Aug. 2015, pp. 195–200.

[59] S. Hahn, M. Protsenko, and T. Müller, "Comparative evaluation of machine learning-based malware detection on Android," in *Proc. Sicherheit*, 2016, pp. 1–10.

[60] K. A. Talha, D. I. Alper, and C. Aydin, "APK Auditor: Permission-based Android malware detection system," *Digit. Invest.*, vol. 13, pp. 1–14, Jun. 2015.

[61] J. G. de la Puerta, B. Sanz, I. Santos, and P. G. Bringas, "Using Dalvik opcodes for malware detection on Android," in *Proc. Int. Conf. Hybrid Artif. Intell. Syst.* Bibao, Spain: Springer, 2015, pp. 416–426.

[62] G. Dai, J. Ge, M. Cai, D. Xu, and W. Li, "SVM-based malware detection for Android applications," in *Proc. WISEC*, 2015, pp. 33:1–33:2.

[63] M. Qiao, A. H. Sung, and Q. Liu, "Merging permission and API features for Android malware detection," in *Proc. Int. Congr. Adv. Appl. Inform. (IIAI)*, 2016, pp. 566–571.

[64] B. Sanz, I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. G. Bringas, "Anomaly detection using string analysis for Android malware detection," in *Proc. Int. Joint Conf. SOCO-CISIS-ICEUTE*. Salamanca, Spain: Springer, 2014, pp. 469–478.

[65] S. Y. Yerima, S. Sezer, G. Mcwilliams, and I. Muttik, "A new Android malware detection approach using Bayesian classification," in *Proc. IEEE Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2013, pp. 121–128.

[66] S. Y. Yerima, S. Sezer, and G. Mcwilliams, "Analysis of Bayesian classification-based approaches for Android malware detection," *IET Inf. Secur.*, vol. 8, no. 1, pp. 25–36, 2014.

[67] Z. Liu, Y. Lai, and Y. Chen, "Android malware detection based on permission combinations," *Int. J. Simul. Process Model.*, vol. 10, no. 4, pp. 315–326, 2015.

[68] Q. Li and X. Li, "Android malware detection based on static analysis of characteristic tree," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discovery*, 2015, pp. 84–91.

[69] M. V. Varsha, P. Vinod, and K. A. Dhanya, "Heterogeneous feature space for Android malware detection," in *Proc. 8th Int. Conf. Contemp. Comput.*, Aug. 2015, pp. 383–388.
[70] K. A. P. da Costa, L. A. da Silva, G. B. Martins, G. H. Rosa, C. R. Pereira, and J. P. Papa, "Malware detection in Android-based mobile environments using optimum-path forest," in *Proc. IEEE Int. Conf. Mach. Learn. Appl.*, Dec. 2016, pp. 754–759.
[71] H. Dong, H. E. Neng-Qiang, H. U. Ge, L. I. Qi, and M. Zhang, "Malware detection method of Android application based on simplification instructions," *J. China Universities Posts Telecommun.*, vol. 21, nos. 23–24, pp. 94–100, 2014.
[72] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan, "Evaluation of Android anti-malware techniques against dalvik bytecode obfuscation," in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Sep. 2015, pp. 414–421.
[73] A. Narayanan, Y. Liu, L. Chen, and J. Liu, "Adaptive and scalable Android malware detection through online learning," in *Proc. IEEE IJCNN*, Jul. 2016, pp. 2484–2491.
[74] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. Usenix Conf. Secur.*, 2013, pp. 527–542.
[75] J. G. de la Puerta, B. Sanz, I. S. Grueiro, and P. G. Bringas, "The evolution of permission as feature for Android malware detection," in *Computational Intelligence in Security for Information Systems* (Advances in Intelligent Systems and Computing), vol. 369. Burgos, Spain: Springer, 2015, pp. 389–400.
[76] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: Analysis of Android intent effectiveness in malware detection," *Comput. Secur.*, vol. 65, pp. 121–134, Mar. 2016.
[77] L. Sun, Z. Li, Q. Yan, W. Srisa-An, and Y. Pan, "SigPID: Significant permission identification for Android malware detection," in *Proc. Int. Conf. Malicious Unwanted Softw.*, 2017, pp. 1–8.
[78] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, "Pindroid: A novel Android malware detection system using ensemble learning methods," *Comput. Secur.*, vol. 68, pp. 36–46, Jul. 2017.
[79] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, Mar. 2016.
[80] N. Mclaughlin, J. M. del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. J. Ahn, "Deep Android malware detection," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 301–308.
[81] S. Feng, "Android security via static program analysis," in *Proc. Workshop MOBISYS Ph.D. Forum*, 2017, pp. 19–20.
[82] S. Hou, A. Saas, L. Chen, Y. Ye, and T. Bourlai, "Deep neural networks for automatic Android malware detection," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining (ASONAM)*, 2017, pp. 803–810. [Online]. Available: http://doi.acm.org/10.1145/3110025.3116211
[83] H. A. Alatwi, T. Oh, E. Fokoue, and B. Stackpole, "Android malware detection using category-based machine learning classifiers," in *Proc. Conf. Inf. Technol. Educ.*, 2016, pp. 54–59.
[84] K.-H.-T. Dam and T. Touili, "Learning Android malware," in *Proc. Int. Conf. Availab., Rel. Secur.*, 2017, p. 59.
[85] A. Ali-Gombe, I. Ahmed, and V. Roussev, "OpSeq: Android malware fingerprinting," in *Proc. Program Protection Reverse Eng. Workshop*, 2015, p. 7.
[86] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in Android apps," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Orlando, FL, USA, Dec. 2017, pp. 288–302. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134642
[87] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "AppHolmes: Detecting and characterizing app collusion among third-party Android markets," in *Proc. Int. Conf. World Wide Web*, 2017, pp. 143–152.
[88] L. Chen, S. Hou, and Y. Ye, "SecureDroid: Enhancing security of machine learning-based detection against adversarial Android malware attacks," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Orlando, FL, USA, Dec. 2017, pp. 362–372. [Online]. Available: http://doi.acm.org/10.1145/3134600.3134636
[89] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[90] X. Chen and S. Zhu, "DroidJust: Automated functionality-aware privacy leakage analysis for Android applications," in *Proc. ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2015, p. 5.
[91] H. Mumtaz and E. S. M. El-Alfy, "Critical review of static taint analysis of Android applications for detecting information leakages," in *Proc. Int. Conf. Inf. Technol.*, 2017, pp. 446–454.
[92] F. Mohsen, H. Bisgin, Z. Scott, and K. Strait, "Detecting Android malwares by mining statically registered broadcast receivers," in *Proc. IEEE 3rd Int. Conf. Collaboration Internet Comput. (CIC)*, Oct. 2017, pp. 67–76.
[93] S. Wu, Y. Zhang, and X. Xiong, "Efficient privacy leakage discovery for Android applications based on static analysis," *Int. J. Hybrid Inf. Technol.*, vol. 9, no. 3, pp. 199–210, 2016.
[94] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: API package-based characterization and detection of mobile ransomware," in *Proc. Symp. Appl. Comput.*, 2017, pp. 1718–1723.
[95] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Poster: Android collusive data leaks with flow-sensitive DIALDroid dataset," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Paris, France, Apr. 2017.
[96] M. Yusof, M. M. Saudi, and F. Ridzuan, "A new mobile botnet classification based on permission and API calls," in *Proc. 7th Int. Conf. Emerg. Secur. Technol. (EST)*, Sep. 2017, pp. 122–127.
[97] W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, "Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers," *Future Gener. Comput. Syst.*, vol. 78, pp. 987–994, Jan. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17300742
[98] X. Liu, J. Liu, W. Wang, Y. He, and X. Zhang, "Discovering and understanding Android sensor usage behaviors with data flow analysis," *World Wide Web*, vol. 21, no. 1, pp. 105–126, 2018.
[99] W. Wang, M. Zhao, and J. Wang, "Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *J. Ambient Intell. Humanized Comput.*, pp. 1–9, Apr. 2018. doi: 10.1007/s12652-018-0803-6.
[100] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of Android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, Jan. 2018. doi: 10.1016/j.neucom.2017.07.030.
[101] X. Fu, D. Lee, and C. Jung, "nAdroid: Statically detecting ordering violations in Android applications," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Vienna, Austria, Feb. 2018, pp. 62–74. [Online]. Available: http://doi.acm.org/10.1145/3168829
[102] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang, "DroidEnsemble: Detecting Android malicious applications with ensemble of string and structural static features," *IEEE Access*, vol. 6, pp. 31798–31807, 2018. doi: 10.1109/ACCESS.2018.2835654.
[103] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "DAPASA: Detecting Android piggybacked apps through sensitive subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 8, pp. 1772–1785, Aug. 2017.
[104] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of Android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, p. 11, 2016.
[105] X. Liu, J. Liu, and W. Wang, "Exploring sensor usage behaviors of Android applications based on data flow analysis," in *Proc. IEEE Int. Perform. Comput. Commun. Conf.*, Dec. 2015, pp. 1–8.
[106] W. Klieber, L. Flynn, W. Snavely, and M. Zheng, "Practical precise taint-flow static analysis for Android app sets," in *Proc. 13th Int. Conf. Availab., Rel. Secur. (ARES)*, Hamburg, Germany, Aug. 2018, pp. 56:1–56:7. doi: 10.1145/3230833.3232825.
[107] P. Vinod, A. Zemmari, and M. Conti, "A machine learning based approach to detect malicious Android apps using discriminant system calls," *Future Gener. Comput. Syst.*, vol. 94, pp. 333–350, May 2019. doi: 10.1016/j.future.2018.11.021.
[108] K. Sharma and B. B. Gupta, "Mitigation and risk factor analysis of Android applications," *Comput. Elect. Eng.*, vol. 71, pp. 416–430, Oct. 2018. doi: 10.1016/j.compeleceng.2018.08.003.
[109] R. Shao, V. Rastogi, Y. Chen, X. Pan, G. Guo, S. Zou, and R. Riley, "Understanding in-app ads and detecting hidden attacks through the mobile app-Web interface," *IEEE Trans. Mobile Comput.*, vol. 17, no. 11, pp. 2675–2688, Nov. 2018. doi: 10.1109/TMC.2018.2809727.
[110] Z. Shan, I. Neamtiu, and R. Samuel, "Self-hiding behavior in Android apps: Detection and characterization," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, Gothenburg, Sweden, May/Jun. 2018, pp. 728–739. doi: 10.1145/3180155.3180214.
[111] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting Android malware leveraging text semantics of network flows," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1096–1109, May 2018. doi: 10.1109/TIFS.2017.2771228.

[112] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018. doi: 10.1109/TIFS.2018.2806891.

[113] F. Shen, J. D. Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek, "Android malware detection using complex-flows," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Atlanta, GA, USA, Jun. 2017, pp. 2430–2437. doi: 10.1109/ICDCS.2017.190.

[114] W. Li, Z. Wang, J. Cai, and S. Cheng, "An Android malware detection approach using weight-adjusted deep learning," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Maui, HI, USA, Mar. 2018, pp. 437–441. doi: 10.1109/ICCNC.2018.8390391.

[115] Z. Chen, Q. Yan, H. Han, S. Wang, L. Peng, L. Wang, and B. Yang, "Machine learning based mobile malware detection using highly imbalanced network traffic," *Inf. Sci.*, vols. 433–434, pp. 346–364, Apr. 2018. doi: 10.1016/j.ins.2017.04.044.

[116] L. C. Navarro, A. K. W. Navarro, A. Grégio, A. Rocha, and R. Dahab, "Leveraging ontologies and machine-learning techniques for malware analysis into Android permissions ecosystems," *Comput. Secur.*, vol. 78, pp. 429–453, Sep. 2018. doi: 10.1016/j.cose.2018.07.013.

[117] N. V. Duc and P. T. Giang, "NADM: Neural network for Android detection malware," in *Proc. 9th Int. Symp. Inf. Commun. Technol. (SoICT)*, Danang, Vietnam, Dec. 2018, pp. 449–455. doi: 10.1145/3287921.3287977.

[118] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee, "Improving accuracy of Android malware detection with lightweight contextual awareness," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 210–221.

[119] Y. Zhang, Y. Yang, and X. Wang, "A novel Android malware detection approach based on convolutional neural network," in *Proc. 2nd Int. Conf. Cryptogr., Secur. Privacy*, 2018, pp. 144–149.

[120] S. Ngamwitroj and B. Limthanmaphon, "Adaptive Android malware signature detection," in *Proc. Int. Conf. Commun. Eng. Technol.*, Feb. 2018, pp. 22–25.

[121] R. Kumar, Z. Xiaosong, R. U. Khan, J. Kumar, and I. Ahad, "Effective and explainable detection of Android malware based on machine learning algorithms," in *Proc. Int. Conf. Comput. Artif. Intell. (ICCAI)*, 2018, pp. 35–40. [Online]. Available: http://doi.acm.org/10.1145/3194452.3194465

[122] J. Jung, J. Choi, S.-J. Cho, S. Han, M. Park, and Y. Hwang, "Android malware detection using convolutional neural networks and data section images," in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, 2018, pp. 149–153. [Online]. Available: http://doi.acm.org/10.1145/3264746.3264780

[123] D. Su, J. Liu, W. Wang, X. Wang, X. Du, and M. Guizani, "Discovering communities of malapps on Android-based mobile cyber-physical systems," *Ad Hoc Netw.*, vol. 80, pp. 104–115, Nov. 2018. doi: 10.1016/j.adhoc.2018.07.015.

[124] A. Machiry, N. Redini, E. Gustafson, Y. Fratantonio, Y. R. Choe, C. Kruegel, and G. Vigna, "Using loops for malware classification resilient to feature-unaware perturbations," in *Proc. 34th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, San Juan, PR, USA, Dec. 2018, pp. 112–123. doi: 10.1145/3274694.3274731.

[125] B. Jung, T. Kim, and E. G. Im, "Malware classification using byte sequence information," in *Proc. Conf. Res. Adapt. Convergent Syst. (RACS)*, Honolulu, HI, USA, Oct. 2018, pp. 143–148. doi: 10.1145/3264746.3264775.

[126] D. Su, J. Liu, X. Wang, and W. Wang, "Detecting Android locker-ransomware on chinese social networks," *IEEE Access*, vol. 7, pp. 20381–20393, 2019. doi: 10.1109/ACCESS.2018.2888568.

[127] M. Z. Mas'ud, S. Sahib, M. F. Abdollah, S. R. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in Android malware detection," in *Proc. Int. Conf. Inf. Sci. Appl. (ICISA)*, May 2014, pp. 1–5.

[128] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[129] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: Automatic security analysis of smartphone applications," in *Proc. 3rd ACM Conf. Data Appl. Secur. Privacy*, 2013, pp. 209–220.

[130] M. R. Amin, M. Zaman, M. S. Hossain, and M. Atiquzzaman, "Behavioral malware detection approaches for Android," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[131] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of Android malware detection based on system calls," in *Proc. ACM Int. Workshop Secur. Privacy Anal.*, 2016, pp. 1–8.

[132] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Acquiring and analyzing app metrics for effective mobile malware detection," in *Proc. ACM Int. Workshop Secur. Privacy Anal.*, 2016, pp. 50–57.

[133] R. S. Pirscoveanu, S. S. Hansen, T. M. Larsen, M. Stevanovic, J. M. Pedersen, and A. Czech, "Analysis of malware behavior: Type classification using machine learning," in *Proc. Int. Conf. Cyber Situational Awareness, Data Anal. Assessment (CyberSA)*, Jun. 2015, pp. 1–7.

[134] H.-S. Ham, H.-H. Kim, M.-S. Kim, and M.-J. Choi, "Linear SVM-based Android malware detection," in *Frontier and Innovation in Future Computing and Communications*. Springer, 2014, pp. 575–585.

[135] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in Android," *J. Parallel Distrib. Comput.*, vol. 103, pp. 22–31, May 2017.

[136] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Comput.*, vol. 20, no. 1, pp. 343–357, 2016.

[137] J. Gajrani, J. Sarswat, M. Tripathi, V. Laxmi, M. S. Gaur, and M. Conti, "A robust dynamic analysis system preventing SandBox detection by Android malware," in *Proc. 8th Int. Conf. Secur. Inf. Netw.*, 2015, pp. 290–295.

[138] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting Android malware using sequences of system calls," in *Proc. 3rd Int. Workshop Softw. Develop. Lifecycle Mobile*, 2015, pp. 13–20.

[139] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying Android malware using dynamically obtained features," *J. Comput. Virol. Hacking Techn.*, vol. 11, no. 1, pp. 9–17, 2015.

[140] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proc. NDSS*, vol. 16, 2016, pp. 21–24.

[141] S.-W. Hsiao, Y. S. Sun, and M. C. Chen, "Behavior grouping of Android malware family," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[142] H. Kurniawan, Y. Rosmansyah, and B. Dabarsyah, "Android anomaly detection system using machine learning classification," in *Proc. Int. Conf. Elect. Eng. Inform. (ICEEI)*, Aug. 2015, pp. 288–293.

[143] J.-W. Jang, J. Yun, J. Woo, and H. K. Kim, "Andro-profiler: Anti-malware system based on behavior profiling of mobile malware," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 737–738.

[144] R. Andriatsimandefitra and V. V. T. Tong, "Capturing Android malware behaviour using system flow graph," in *Proc. Int. Conf. Netw. Syst. Secur.* New York, NY, USA: Springer, 2014, pp. 534–541.

[145] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "Droidscribe: Classifying Android malware based on runtime behavior," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2016, pp. 252–261.

[146] M. Zaman, T. Siddiqui, M. R. Amin, and M. S. Hossain, "Malware detection in Android by network traffic analysis," in *Proc. Int. Conf. Netw. Syst. Secur. (NSysS)*, Jan. 2015, pp. 1–5.

[147] R. Andriatsimandefitra and V. V. T. Tong, "Detection and identification of Android malware based on information flow monitoring," in *Proc. IEEE 2nd Int. Conf. Cyber Secur. Cloud Comput. (CSCloud)*, Nov. 2015, pp. 200–203.

[148] V. Wahanggara and Y. Prayudi, "Malware detection through call system on Android smartphone using vector machine method," in *Proc. 4th Int. Conf. Cyber Secur., Cyber Warfare, Digit. Forensic (CyberSec)*, Oct. 2015, pp. 62–67.

[149] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. 22nd Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2015, pp. 1–15. [Online]. Available: https://www.ndss-symposium.org/ndss2015/copperdroid-automatic-reconstruction-android-malware-behaviors

[150] Y. J. Ham and H.-W. Lee, "Detection of malicious Android mobile applications based on aggregated system call events," *Int. J. Comput. Commun. Eng.*, vol. 3, no. 2, p. 149, 2014.

[151] Y. J. Ham, D. Moon, H.-W. Lee, J. D. Lim, and J. N. Kim, "Android mobile application system call event pattern analysis for determination of malicious attack," *Int. J. Secur. Appl.*, vol. 8, no. 1, pp. 231–246, 2014.

[152] A. Feizollah, N. B. Anuar, R. Salleh, and F. Amalina, "Comparative study of k-means and mini batch k-means clustering algorithms in Android malware detection using network traffic analysis," in *Proc. Int. Symp. Biometrics Secur. Technol. (ISBAST)*, Aug. 2014, pp. 193–197.

[153] H. Yang and R. Tang, "Power consumption based Android malware detection," *J. Elect. Comput. Eng.*, vol. 2016, Mar. 2016, Art. no. 6860217. doi: 10.1155/2016/6860217.

[154] O. S. Jarquín, U. Zurutuza, R. Uribeetxeberria, L. Delosières, and S. Nadjm-Tehrani, "Detection and visualization of Android malware behavior," *J. Elect. Comput. Eng.*, vol. 2016, Feb. 2016, Art. no. 8034967. doi: 10.1155/2016/8034967.
[155] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly, "PowerSpy: Location tracking using mobile device power analysis," in *Proc. USENIX Secur. Symp.*, 2015, pp. 785–800.
[156] N. Artenstein and I. Revivo, "Man in the binder: He who controls IPC, controls the droid," in *Proc. Black Hat*, 2014, pp. 1–23.
[157] M. Salehi, F. Daryabar, and M. H. Tadayon, "Welcome to binder: A kernel level attack model for the binder in Android operating system," in *Proc. 8th Int. Symp. Telecommun. (IST)*, Sep. 2016, pp. 156–161.
[158] V. G. Shankar, G. Somani, M. S. Gaur, V. Laxmi, and M. Conti, "Andro-Taint: An efficient Android malware detection framework using dynamic taint analysis," in *Proc. ISEA IEEE Asia Secur. Privacy (ISEASP)*, Jan. 2017, pp. 1–13.
[159] H. Ruan, X. Fu, X. Liu, X. Du, and B. Luo, "Analyzing Android application in real-time at kernel level," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2017, pp. 1–9.
[160] T. Bhatia and R. Kaushal, "Malware detection in Android based on dynamic analysis," in *Proc. Int. Conf. Cyber Secur. Protection Digit. Services (Cyber Secur.)*, Jun. 2017, pp. 1–6.
[161] H. Cai and B. G. Ryder, "Understanding Android application programming and security: A dynamic study," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 364–375.
[162] J. Schüette, A. Küechler, and D. Titze, "Practical application-level dynamic taint analysis of Android apps," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Aug. 2017, pp. 17–24.
[163] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "TaintMan: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices," *IEEE Trans. Dependable Secure Comput.*, to be published.
[164] S. Zhang and X. Xiao, "CSCdroid: Accurately detect Android malware via contribution-level-based system call categorization," in *Proc. IEEE Trustcom/BigDataSE/ICESS*, Aug. 2017, pp. 193–200.
[165] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Improving dynamic analysis of Android apps using hybrid test input generation," in *Proc. Int. Conf. Cyber Secur. Protection Digit. Services (Cyber Secur.)*, Jun. 2017, pp. 1–8.
[166] D. Sun, C. Guo, D. Zhu, and W. Feng, "Secure HybridApp: A detection method on the risk of privacy leakage in HTML5 hybrid applications based on dynamic taint tracking," in *Proc. 2nd IEEE Int. Conf. Comput. Commun. (ICCC)*, Oct. 2016, pp. 2771–2775.
[167] Q. Tang, W. Zhang, X. Li, and B. Wang, "X-Prcaf: Xposed based protecting cache file from leaks in Android social applications," in *Proc. 3rd Int. Conf. Trustworthy Syst. Appl. (TSA)*, Sep. 2016, pp. 17–22.
[168] F. Martinelli, F. Marulli, and F. Mercaldo, "Evaluating convolutional neural network for effective mobile malware detection," in *Proc. 21st Int. Conf. Knowl.-Based Intell. Inf. Eng. Syst. (KES)*, Marseille, France, Sep. 2017, pp. 2372–2381. doi: 10.1016/j.procs.2017.08.216.
[169] W. Fan, Y. Sang, D. Zhang, R. Sun, and Y. Liu, "DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications," *Comput. Secur.*, vol. 70, pp. 224–237, Sep. 2017. doi: 10.1016/j.cose.2017.06.001.
[170] E. B. Karbab, M. Debbabi, S. Alrabaee, and D. Mouheb, "DySign: Dynamic fingerprinting for the automatic detection of Android malware," *CoRR*, vol. abs/1702.05699, pp. 1–8, Feb. 2017. [Online]. Available: http://arxiv.org/abs/1702.05699
[171] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: Android malware detection using machine learning," in *Proc. 3rd ACM Int. Workshop Secur. Privacy Anal.*, 2017, pp. 65–72.
[172] L. Chen, M. Zhang, C.-Y. Yang, and R. Sahita, "POSTER: Semi-supervised classification for dynamic Android malware detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2479–2481.
[173] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "DynaLog: An automated dynamic analysis framework for characterizing Android applications," in *Proc. Int. Conf. IEEE Cyber Secur. Protection Digit. Services (Cyber Secur.)*, Jun. 2016, pp. 1–8.
[174] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4MalDroid: A deep learning framework for Android malware detection based on linux kernel system call graphs," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Workshops (WIW)*, Oct. 2016, pp. 104–111.
[175] H. Papadopoulos, N. Georgiou, C. Eliades, and A. Konstantinidis, "Android malware detection with unbiased confidence guarantees," *Neurocomputing*, vol. 280, pp. 3–12, Mar. 2018.
[176] J. M. Vidal, M. A. S. Monge, and L. J. García-Villalba, "A novel pattern recognition system for detecting Android malware by analyzing suspicious boot sequences," *Knowl.-Based Syst.*, vol. 150, pp. 198–217, Jun. 2018. doi: 10.1016/j.knosys.2018.03.018.
[177] B. Li, Y. Zhang, J. Li, W. Yang, and D. Gu, "AppSpear: Automating the hidden-code extraction and reassembling of packed Android malware," *J. Syst. Softw.*, vol. 140, pp. 3–16, Jun. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218300311
[178] A. Mahindru and P. Singh, "Dynamic permissions based Android malware detection using machine learning techniques," in *Proc. 10th Innov. Softw. Eng. Conf.*, 2017, pp. 202–210.
[179] D. Wu, Y. Cheng, D. Gao, Y. Li, and R. H. Deng, "SCLib: A practical and lightweight defense against component hijacking in Android applications," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, Tempe, AZ, USA, Mar. 2018, pp. 299–306. [Online]. Available: http://doi.acm.org/10.1145/3176258.3176336
[180] Y. Yang, Z. Wei, Y. Xu, H. He, and W. Wang, "DroidWard: An effective dynamic analysis method for vetting Android applications," *Cluster Comput.*, vol. 21, no. 1, pp. 265–275, 2016.
[181] G. Dini, F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, and D. Sgandurra, "Risk analysis of Android applications: A user-centric solution," *Future Gener. Comput. Syst.*, vol. 80, pp. 505–518, Mar. 2018. doi: 10.1016/j.future.2016.05.035.
[182] H. Cai, N. Meng, B. G. Ryder, and D. Yao, "DroidCat: Effective Android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 6, pp. 1455–1470, 2019. doi: 10.1109/TIFS.2018.2879302.
[183] A. Martín, V. Rodríguez-Fernández, and D. Camacho, "CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains," *Eng. Appl. Artif. Intell.*, vol. 74, pp. 121–133, Sep. 2018. doi: 10.1016/j.engappai.2018.06.006.
[184] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018. doi: 10.1109/TDSC.2016.2536605.
[185] A. Muñoz, I. Martín, A. Guzmán, and J. A. Hernández, "Android malware detection from Google Play meta-data: Selection of important features," in *Proc. CNS*, Sep. 2015, pp. 701–702.
[186] K. Wang, T. Song, and A. Liang, "Mmda: Metadata based malware detection on Android," in *Proc. Int. Conf. Comput. Intell. Secur.*, Dec. 2017, pp. 598–602.
[187] A. Martín, A. Calleja, H. D. Menéndez, J. E. Tapiador, and D. Camacho, "ADROIT: Android malware detection using meta-information," in *Proc. IEEE Symp. Comput. Intell. (SSCI)*, Athens, Greece, Dec. 2016, pp. 1–8. doi: 10.1109/SSCI.2016.7849904.
[188] N. B. Akhuseyinoglu and K. Akhuseyinoglu, "AntiWare: An automated Android malware detection tool based on machine learning approach and official market metadata," in *Proc. IEEE 7th Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, New York, NY, USA, Oct. 2016, pp. 1–7. doi: 10.1109/UEMCON.2016.7777867.
[189] J. Wu, M. Yang, and T. Luo, "PACS: Pemission abuse checking system for Android applictions based on review mining," in *Proc. IEEE Conf. Dependable Secure Computing*, Aug. 2017, pp. 251–258.
[190] I. Gurulian, K. Markantonakis, L. Cavallaro, and K. Mayes, "Reprint of 'you can't touch this: Consumer-centric Android application repackaging detection,'" *Future Gener. Comput. Syst.*, vol. 80, pp. 537–545, Mar. 2018. doi: 10.1016/j.future.2017.11.011.
[191] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012. doi: 10.1007/s10844-010-0148-x.
[192] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS*, Feb. 2012, vol. 25. no. 4, pp. 50–52.
[193] W. C. Wu and S. H. Hung, "DroidDolphin: A dynamic Android malware detection framework using big data and machine learning," in *Proc. Conf. Res. Adapt. Convergent Syst.*, 2014, pp. 247–252.
[194] S. Bhandari, R. Gupta, V. Laxmi, M. S. Gaur, A. Zemmari, and M. Anikeev, "DRACO: DRoid analyst combo an Android malware analysis framework," in *Proc. Int. Conf. Secur. Inf. Netw.*, 2015, pp. 283–289.
[195] T. Vidas, J. Tan, J. Nahata, C. L. Tan, P. Tague, and P. Tague, "A5: Automated analysis of adversarial Android applications," in *Proc. ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2014, pp. 39–50.

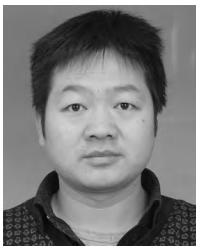
[196] S. Chen, Z. Tang, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A streaminglized machine learning-based system for detecting Android malware," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 377–388.
[197] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: Android malware characterization and detection using deep learning," *Tsinghua Sci. Technol.*, vol. 21, no. 1, pp. 114–123, Feb. 2016.
[198] M. Ozdemir and I. Sogukpinar, "An Android malware detection architecture based on ensemble learning," *Trans. Mach. Learn. Artif. Intell.*, vol. 2, no. 3, pp. 90–106, 2014.
[199] A. Mohaisen, O. Alrawi, and M. Mohaisen, "AMAL: High-fidelity, behavior-based automated malware analysis and classification," *Comput. Secur.*, vol. 52, pp. 251–266, Jul. 2015.
[200] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng, "Attack tree based Android malware detection with hybrid analysis," in *Proc. IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Sep. 2015, pp. 380–387.
[201] X. Wang, Y. Yang, and Y. Zeng, "Accurate mobile malware detection and classification in the cloud," *SpringerPlus*, vol. 4, no. 1, p. 583, 2015.
[202] P. Singh, P. Tiwari, and S. Singh, "Analysis of malicious behavior of Android apps," *Procedia Comput. Sci.*, vol. 79, pp. 215–220, Jan. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916001599
[203] J. Malik and R. Kaushal, "CREDROID: Android malware detection by network traffic analysis," in *Proc. ACM Workshop Privacy-Aware Mobile Comput.*, 2016, pp. 28–36.
[204] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of Android malware," in *Proc. Eur. Workshop Syst. Secur.*, 2014, p. 5.
[205] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann, "Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques," *Int. J. Inf. Secur.*, vol. 14, no. 2, pp. 141–153, 2015.
[206] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on Android," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 915–930.
[207] P. Wang and Y.-S. Wang, "Malware behavioural detection and vaccine development by using a support vector model classifier," *J. Comput. Syst. Sci.*, vol. 81, no. 6, pp. 1012–1026, 2015.
[208] D. Quan, L. Zhai, F. Yang, and P. Wang, "Detection of Android malicious apps based on the sensitive behaviors," in *Proc. TrustCom*, Sep. 2014, pp. 877–883.
[209] Y. Zhang, M. Yang, Z. Yang, G. Gu, P. Ning, and B. Zang, "Permission use analysis for vetting undesirable behaviors in Android apps," *IEEE Trans. Inf. Forensics Security*, vol. 9, no. 11, pp. 1828–1842, Nov. 2014.
[210] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy," in *Proc. Symp. Netw. Distrib. Syst. Secur.*, 2016, pp. 1–15.
[211] M. Rapoport, P. Suter, E. Wittern, O. Lhóták, and J. Dolby, "Who you gonna call?: Analyzing Web requests in Android applications," in *Proc. IEEE/ACM Int. Conf. Mining Softw. Repositories*, May 2017, pp. 80–90.
[212] Z. Lv, Z. Guo, and C. Chen, "Research on Android intent security detection based on machine learning," in *Proc. Int. Conf. Inf. Sci. Control Eng.*, Jul. 2017, pp. 569–574.
[213] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "MR-Droid: A scalable and prioritized analysis of inter-app communication risks," in *Proc. IEEE Secur. Privacy Workshops*, May 2017, pp. 189–198.
[214] M. Leeds, M. Keffeler, and T. Atkison, "A comparison of features for Android malware detection," in *Proc. ACM Southeast Regional Conf.*, 2017, pp. 63–68.
[215] T. Yang, H. Cui, and S. Niu, "Dynamic loading vulnerability detection for Android applications through ensemble learning," *Chin. J. Electron.*, vol. 26, no. 5, pp. 960–965, 2017.
[216] F. Yang, Y. Zhuang, and J. Wang, "Android malware detection using hybrid analysis and machine learning technique," in *Proc. Int. Conf. Cloud Comput. Secur.* Nanjing, China: Springer, 2017, pp. 565–575.
[217] Z.-U. Rehman, S. N. Khan, K. Muhammad, J. W. Lee, Z. Lv, S. W. Baik, P. A. Shah, K. Awan, and I. Mehmood, "Machine learning-assisted signature and heuristic-based detection of malwares in Android devices," *Comput. Elect. Eng.*, vol. 69, pp. 828–841, Jun. 2018. doi: 10.1016/j.compeleceng.2017.11.028.
[218] X. Wang, Y. Yang, Y. Zeng, C. Tang, J. Shi, and K. Xu, "A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection," in *Proc. Int. Workshop*, 2015, pp. 15–22.
[219] X. Wang, S. Zhu, D. Zhou, and Y. Yang, "Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of Android malware," in *Proc. Comput. Secur. Appl. Conf.*, 2017, pp. 350–361.
[220] A. T. Kabakus and I. A. Dogru, "An in-depth analysis of Android malware using hybrid techniques," *Digit. Invest.*, vol. 24, pp. 25–33, Mar. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287617303183
[221] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, "SAMADroid: A novel 3-level hybrid malware detection model for Android operating system," *IEEE Access*, vol. 6, pp. 4321–4339, 2018. doi: 10.1109/ACCESS.2018.2792941.
[222] A. I. Ali-Gombe, B. Saltaformaggio, J. Ramanujam, D. Xu, and G. G. Richard, III, "Toward a more dependable hybrid analysis of Android malware using aspect-oriented programming," *Comput. Secur.*, vol. 73, pp. 235–248, Mar. 2018. doi: 10.1016/j.cose.2017.11.006.
[223] Z. Rehman, S. N. Khan, K. Muhammad, J. W. Lee, Z. Lv, S. W. Baik, P. A. Shah, K. Awan, and I. Mehmood, "Machine learning-assisted signature and heuristic-based detection of malwares in Android devices," *Comput. Elect. Eng.*, vol. 69, pp. 828–841, Jul. 2018. doi: 10.1016/j.compeleceng.2017.11.028.
[224] A. Arora, S. K. Peddoju, V. Chouhan, and A. Chaudhary, "Poster: Hybrid Android malware detection by combining supervised and unsupervised learning," in *Proc. 24th Annu. Int. Conf. Mobile Comput. Netw.*, 2018, pp. 798–800.
[225] J. Liu, D. Wu, and J. Xue, "TDroid: Exposing app switching attacks in Android with control flow specialization," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 236–247. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238188
[226] J. Jiang, S. Li, M. Yu, K. Chen, C. Liu, W. Huang, and G. Li, "MRDroid: A multi-act classification model for Android malware risk assessment," in *Proc. 15th IEEE Int. Conf. Mobile Ad Hoc Sensor Syst. (MASS)*, Chengdu, China, Oct. 2018, pp. 64–72. doi: 10.1109/MASS.2018.00020.
[227] M. Kakavand, M. Dabbagh, and A. Dehghantanha, "Application of machine learning algorithms for Android malware detection," in *Proc. CIIS*, Nov. 2018, pp. 32–36.
[228] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the Android ecosystem," *IEEE Trans. Mobile Comput.*, to be published. doi: 10.1109/TMC.2019.2903186.
[229] X. Liu, J. Liu, W. Wang, and S. Zhu, "Android single sign-on security: Issues, taxonomy and directions," *Future Gener. Comput. Syst.*, vol. 89, pp. 402–420, Dec. 2018.
[230] H. Kang, A. Mohaisen, A. Mohaisen, and H. K. Kim, "Detecting and classifying Android malware using static analysis along with creator information," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, 2015, Art. no. 479174.
[231] S. Feldman, D. Stadther, and B. Wang, "Manilyzer: Automated Android malware detection through manifest analysis," in *Proc. IEEE Int. Conf. Mobile Ad Hoc Sensor Syst.*, Oct. 2015, pp. 767–772.
[232] X. Yang, D. Lo, L. Li, X. Xia, T. F. Bissyandé, and J. Klein, "Characterizing malicious Android apps by mining topic-specific data flow signatures," *Inf. Softw. Technol.*, vol. 90, pp. 27–39, Oct. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095058491730366X
[233] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 309–320.
[234] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen, "A pragmatic Android malware detection procedure," *Comput. Secur.*, vol. 70, pp. 689–701, Sep. 2017.
[235] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 422–433.
[236] J. W. Jang, H. Kang, J. Woo, A. Mohaisen, and H. K. Kim, "Andro-Dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information," *Comput. Secur.*, vol. 58, pp. 125–138, May 2016. doi: 10.1016/j.eswa.2017.04.003.
[237] F. Martinelli, F. Mercaldo, and A. Saracino, "BRIDEMAID: An hybrid tool for accurate detection of Android malware," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 899–901.
[238] J. Lin, X. Zhao, and H. Li, "Target: Category-based Android malware detection revisited," in *Proc. ACSW*, 2017, pp. 74:1–74:9.

[239] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *Proc. Conf. Centre Adv. Stud. Collaborative Res. (CASCON)*, 1999, pp. 1–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008
[240] E. Bodden, "Inter-procedural data-flow analysis with IFDS/IDE and soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 3–8.
[241] M. Ahmad, V. Costamagna, B. Crispo, and F. Bergadano, "TeICC: Targeted execution of inter-component communications in Android," in *Proc. Symp. Appl. Comput.*, 2017, pp. 1747–1752.
[242] W. Wang, J. Liu, G. Pitsilis, and X. Zhang, "Abstracting massive data for lightweight intrusion detection in computer networks," *Inf. Sci.*, vols. 433–434, pp. 417–430, Apr. 2018.
[243] W. Wang, T. Guyet, R. Quiniou, M. Cordier, F. Masseglia, and X. Zhang, "Autonomic intrusion detection: Adaptively detecting anomalies over unlabeled audit data streams in computer networks," *Knowl.-Based Syst.*, vol. 70, pp. 103–117, Nov. 2014. doi: 10.1016/j.knosys.2014.06.018.
[244] W. Wang, Y. He, J. Liu, and S. Gombault, "Constructing important features from massive network traffic for lightweight intrusion detection," *IET Inf. Secur.*, vol. 9, no. 6, pp. 374–379, 2015. doi: 10.1049/iet-ifs.2014.0353.
[245] W. Wang, X. Guan, and X. Zhang, "A novel intrusion detection method based on principle component analysis in computer security," in *Proc. Int. Symp. Neural Netw., Adv. Neural Netw.*, Dalian, China, Aug. 2004, pp. 657–662. doi: 10.1007/978-3-540-28648-6_105.
[246] W. Wang and X. Zhang, "High-speed web attack detection through extracting exemplars from HTTP traffic," in *Proc. ACM Symp. Appl. Comput. (SAC)*, TaiChung, Taiwan, Mar. 2011, pp. 1538–1543. doi: 10.1145/1982185.1982512.
[247] W. Wang and R. Battiti, "Identifying intrusions in computer networks with principal component analysis," in *Proc. 1st Int. Conf. Availab., Rel. Secur. (ARES)*, Apr. 2006, pp. 270–279. doi: 10.1109/ARES.2006.73.
[248] X. Guan, W. Wang, and X. Zhang, "Fast intrusion detection based on a non-negative matrix factorization model," *J. Netw. Comput. Appl.*, vol. 32, no. 1, pp. 31–44, 2009. doi: 10.1016/j.jnca.2008.04.006.
[249] W. Wang, X. Zhang, and S. Gombault, "Constructing attribute weights from computer audit data for effective intrusion detection," *J. Syst. Softw.*, vol. 82, no. 12, pp. 1974–1981, 2009. doi: 10.1016/j.jss.2009.06.040.
[250] W. Wang, X. Guan, X. Zhang, and L. Yang, "Profiling program behavior for anomaly intrusion detection based on the transition and frequency property of computer audit data," *Comput. Secur.*, vol. 25, no. 7, pp. 539–550, 2006. doi: 10.1016/j.cose.2006.05.005.
[251] W. Wang, X. Zhang, S. Gombault, and S. J. Knapskog, "Attribute normalization in network intrusion detection," in *Proc. 10th Int. Symp. Pervas. Syst., Algorithms, Netw. (ISPAN)*, Kaohsiung, Taiwan, Dec. 2009, pp. 448–453. doi: 10.1109/I-SPAN.2009.49.
[252] W. Wang, X. Guan, and X. Zhang, "Processing of massive audit data streams for real-time anomaly intrusion detection," *Comput. Commun.*, vol. 31, no. 1, pp. 58–72, 2008. doi: 10.1016/j.comcom.2007.10.010.
[253] C. Zhang, C. Liu, X. Zhang, and G. Almpanidis, "An up-to-date comparison of state-of-the-art classification algorithms," *Expert Syst. Appl.*, vol. 82, pp. 128–150, Oct. 2017.
[254] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2017.
[255] V. N. Cooper, H. Shahriar, and H. M. Haddad, "A survey of Android malware characterisitics and mitigation techniques," in *Proc. Int. Conf. Inf. Technol., New Gener.*, Apr. 2014, pp. 327–332.
[256] B. Rashidi and C. J. Fung, "A survey of Android security threats and defenses," *JoWUA*, vol. 6, no. 3, pp. 3–35, 2015.
[257] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Proc. 14th Int. Conf. Secur. Privacy Commun. Netw. (SecureComm)*, Singapore, Aug. 2018, pp. 172–192.
[258] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo, "Detection of obfuscation techniques in Android applications," in *Proc. 13th Int. Conf. Availab., Rel. Secur. (ARES)*, Hamburg, Germany, Aug. 2018, pp. 57:1–57:9. doi: 10.1145/3230833.3232823.
[259] O. Mirzaei, J. M. de Fuentes, J. E. Tapiador, and L. González-Manzano, "AndrODet: An adaptive Android obfuscation detector," *Future Gener. Comput. Syst.*, vol. 90, pp. 240–261, Jan. 2019. doi: 10.1016/j.future.2018.07.066.

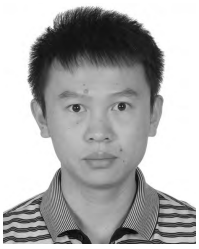
WEI WANG received the Ph.D. degree in control science and engineering from Xi'an Jiaotong University, in 2006. He was a Postdoctoral Researcher with the University of Trento, Italy, from 2005 to 2006. He was a Postdoctoral Researcher with TELECOM Bretagne, and also with INRIA, France, from 2007 to 2008. He is currently a Full Professor with the School of Computer and Information Technology, Beijing Jiaotong University, China. He is an Editorial Board member of Computers & Security and a Young AE of the Frontiers of Computer Science. He has authored or coauthored over 80 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer, and network security. He was a European ERCIM Fellow of the Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009 to 2011.

MEICHEN ZHAO received the B.S. degree from Shanxi University, China, in 2016. She is currently pursuing the M.S. degree with the School of Computer and Information Technology, Beijing Jiaotong University, China. Her main research interest includes mobile security.

ZHENZHEN GAO received the B.S. degree from Hebei University, China, in 2016. She is currently pursuing the M.S. degree with the School of Computer and Information Technology, Beijing Jiaotong University, China. Her main research interest includes mobile security.



GUANGQUAN XU received the Ph.D. degree from Tianjin University, in 2008. He is currently a Ph.D. and a Full Professor with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, China. He is also the Director of the Network Security Joint Lab and the Network Attack and Defense Joint Lab. His research interests include cybersecurity and trust management. He has published over 70 papers in reputable international journals and conferences, including the IEEE IoT J, FGCS, IEEE access, PUC, JPDC, and IEEE multimedia. He is a member of the CCF. He has served as a TPC member for the IEEE UIC 2018, SPNCE2019, IEEE UIC2015, IEEE ICECCS 2014, and a Reviewer for journals, including IEEE Access, ACM TIST, JPDC, IEEE TITS, soft computing, FGCS, and *Computational Intelligence*.



HEQUN XIAN received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, in 2009. He was a Visiting Scholar with the College of Information Science and Technology, Pennsylvania State University. His research interests include cryptography, cloud computing security, and network security.

YUANYUAN LI received the B.S. and M.S. degrees from Beijing Jiaotong University, China, in 2015 and 2018, respectively. Her main research interest includes mobile security.

XIANGLIANG ZHANG received the Ph.D. degree in computer science from INRIA-University Paris-Sud 11, France, in 2010. She is currently an Associate Professor and directs the Machine Intelligence and Knowledge Engineering (MINE) Laboratory, Division of Computer, Electrical and Mathematical Sciences and Engineering, King Abdullah University of Science and Technology (KAUST). She has authored or coauthored over 100 refereed papers in various journals and conferences. Her main research interests and experiences are in diverse areas of machine intelligence, and knowledge engineering and their applications, such as information security and privacy.