Data-Driven Android Malware Intelligence: A Survey

Junyang Qiu[1(✉)], Surya Nepal[2], Wei Luo[1], Lei Pan[1], Yonghang Tai[3], Jun Zhang[4], and Yang Xiang[4]

[1] School of Information Technology, Deakin University, Melbourne, Australia
{qiuju,wei.luo,l.pan}@deakin.edu.au
[2] Data61, CSIRO, Sydney, Australia
surya.nepal@data61.csiro.au
[3] School of Physics and Electronic Information, Yunnan Normal University, Kunming, China
taiyonghang@ynnu.edu.cn
[4] School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia
{junzhang,yxiang}@swin.edu.au

Abstract. Android has dominated the smartphone market and become the most popular mobile operating system. This rapidly increasing market share of Android has contributed to the boom of Android malware in numbers and in varieties. There exist many techniques which are proposed to accurately detect malware, e.g., software engineering-based techniques and machine learning (ML)-based techniques. In this paper, our main contributions are threefold: We reviewed the existing analysis techniques for Android malware detection; We focused on the code analysis based detection techniques under the ML frameworks; We gave the future research challenges and directions about Android malware analysis.

Keywords: Android malware detection · Static analysis · Dynamic analysis · Hybrid analysis · Feature extraction · Machine learning · Code obfuscation

1 Introduction

The Android mobile devices continue to dominate the global mobile market, with about 86.8% market share in the third quarter of 2018 according to the statistical information published by *IDC Corporate*[1]. Almost eight out of ten people worldwide use an Android mobile phone because they are cheap to buy[2]. Android has become the most popular operating system without a doubt. Due to the fact that Android is an open source operating system, thus users can easily

[1] https://www.idc.com/promo/smartphone-market-share/os.
[2] https://www.gdatasoftware.com/.

X. Chen et al. (Eds.): ML4CS 2019, LNCS 11806, pp. 183–202, 2019.
https://doi.org/10.1007/978-3-030-30619-9_14

download and install a wide variety of applications from both official (*Google Play*[3]) and third-party (e.g., *WanDouJia*[4], *AnZhi*[5]) app stores (Currently there are approximately 2.6 million Android apps available at *Google Play*[6]). However, along with Android's popularity and its openness, Android mobile device users have become the most attractive targets of cyber criminals as the number of malicious apps has skyrocketed at an alarming rate. Figure 1 presents the number of Android malware samples being detected per year from 2012 to 2018[7]. It is estimated that almost 12,000 new Android malware samples being detected per day in 2018. Besides, the number of Android malware families has reached about 1,200 [62]. In addition, the sophisticated Android malware samples may be implemented with various strategies (e.g., code obfuscation, encryption) to evade detection.

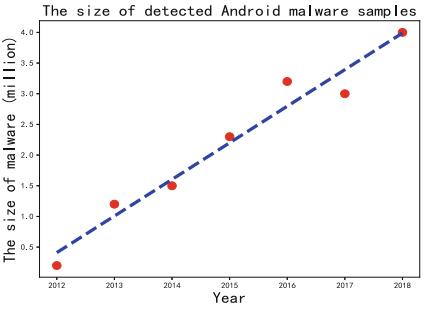

Fig. 1. The number of Android malware samples detected per year from 2012 to 2018.

To preserve a clean and safe ecosystem for Android users, both the academic researchers and the security vendors have invested enormous effort to design effective techniques to defend against Android malware samples or further categorize them into specific malware families [5,25,27,32,42,47,52,83,86,87]. Generally, the existing techniques for malware detection can be roughly divided into three categories [6]. The first one is called static analysis technique, which inspects the disassembled source code to find any potential suspicious functionalities without executing the application. The second one is the dynamic analysis technique, also called behaviors analysis technique. Dynamic analysis executes the given application in an isolated environment (e.g., sandbox, simulator, virtual machine), then monitors and traces its behaviors. The combination of static analysis and dynamic analysis is the third category called hybrid analysis technique [6].

[3] https://play.google.com/store.
[4] https://www.wandoujia.com/.
[5] http://www.anzhi.com/.
[6] https://www.appbrain.com/stats/number-of-android-apps.
[7] https://www.gdatasoftware.com/.

To provide a detailed review about Android malware detection, in this paper, our contributions are threefold: Firstly, we reviewed the existing Android malware detection techniques (including static, dynamic and hybrid techniques) as well as the advantages and disadvantages of each technique. Secondly, in the defender's perspective, targeting the Android code analysis, we introduced the machine learning based Android malware detection framework. We provided an overview of the framework, and then the involved techniques and challenges were reviewed in detail. In addition, we share our views of future potential research directions about the Android malware analysis.

The remaining of this paper is structured as follows. Section 2 presented the research status of Android malware detection. In Sect. 3, the Machine Learning framework for Android malware detection was reviewed. The future research direction and conclusion about this paper were given in Sects. 4 and 5, respectively.

2 Traditional Software Engineering Based Android Malware Analysis

A large number of Android malware analysis methods are built on traditional software engineering technique. Generally, software engineering technique can be roughly divided into three categories: Static code analysis, dynamic behavior analysis, and hybrid analysis. In this Section, we briefly review these three categories.

2.1 Static Code Analysis

The static code analysis is performed by disassembling and analyzing the source code of the given Android applications without executing it [79]. The static code analysis can be further categorized into signature based technique, permission based technique, the Dalvik bytecode-based technique, and the hybrid static analysis technique.

Signature-Based Technique

The signature involved methods are high efficiency and have been widely used by commercial malware detection products. The key building block of signature technique is to generate robust and accurate signatures based on the specific strings or semantic patterns in the source code [26]. Zheng et al. [85] designed DroidAnalytics, a signature-based analysis system which automatically collected Android malware samples, produced signatures, retrieved the information, and associated the malware samples based on a similarity score. Feng et al. proposed Apposcopy, a novel semantics-based approach for detecting a common class of Android malware samples that steals users' privacy data [26]. In [27], Feng et al. further implemented ASTROID, a system for automatically generating semantic Android malware signatures from very few malicious samples within a malware

family. The core idea underlying ASTROID was to look for a Maximally Suspicious Common Subgraph that was shared between all the known malicious samples within an Android malware family [37].

Permission-Based Technique

To ensure the security of the Android operating system, the permission management plays an indispensable role in governing the access privilege [23]. The software authors must declare the requested permissions in the *AndroidManifest.xml* file. Thus, the core idea of permission-based technique is focusing on analyzing the requested sensitive or suspicious permissions to identify the potential malware samples. In 2009, Enck et al. proposed Kirin as a security service for Android analysis [22]. Without complicated and boring code inspection process, Kirin provided practical light-weight certification of Android applications at the installation time using the meaningful security rules to hinder malware samples. ASEDS was created using the Security Distance model to evaluate the risk level of specific combination of permissions [67]. Wu et al. provided a static analysis to extract the permissions related to the APT call traces from *AndroidManifest.xml* [74]. In 2013, PUMA was designed to perform Android malware detection using the permission usage features [57].

Dalvik Bytecode-Based Technique

Android software is usually developed using Java and compiled into Java bytecode. To execute more efficiently, the Java bytecode is optimized to Dalvik bytecode *classes.dex*. The *classes.dex* bytecode contains abundant semantic information, e.g., API calls, data flows, which is related to the application behaviors. The main idea of Dalvik bytecode-based technique is to disassemble the binary code and then analyze the source code to identify the Android malicious samples. A significant tool was Soot[8] originally designed by the Sable Group of McGill University. Soot can translate the Android applications into several intermediate representations, such as *Baf*, *jimple*, *Shimple*, and *Grimp*. An improved version of Soot, called Dexpler was presented in [10]. A robust and light-weight system called DroidAPIMiner was implemented to detect Android malware [1]. DroidAPIMiner extracted the API related semantic information (such as critical API calls, their package level information, and parameters) within the bytecode to represent Android samples. In 2017, HinDroid was proposed using the structured heterogeneous information network to represent the Android applications [32]. An approach named MaMaDroid was presented to detect Android malware by modeling the sequences of API calls as Markov chains [42].

Hybrid Static Analysis Technique

Some works have been conducted to extend the hybrid static analysis by analyzing both the *AndroidManifest.xml* file as well as disassembled *classes.dex* code. In [58], Sato et al. parsed various types of features (including permissions, intent filters, process names and the number of redefined permissions) to characterize

[8] https://sable.github.io/soot/.

the pattern of Android malicious samples. In 2014, Arp et al. proposed a light-weight method named Drebin to detect Android malware samples directly on the device [5]. Drebin extracted 4 types of feature sets from *AndroidManifest.xml* and other 4 feature sets from disassembled *classes.dex* files to characterize the Android applications. Arzt et al. designed a novel and accurate static taint analysis tool named FlowDroid in [7]. Different from the previous approaches, to reduce the false alarm rate, FlowDroid modeled Android's lifecycle or callback methods.

In summary, the static analysis techniques are efficient since they target the source code of the Android software. However, an increasing number of Android malicious samples have been obfuscated or encrypted using various tricks to evade detection [38,66,79]. Under this circumstance, it is difficult to disassemble the binary bytecode and detect the malicious samples accurately. Besides, the static analysis will overestimate the code execution paths. In addition, static analysis techniques are often accompanied by high false positive rate.

2.2 Dynamic Behavior Analysis

Dynamic behavior analysis is conducted by monitoring and tracing the behaviors of Android application during the execution to determine whether it is malicious or not [15].

In 2014, an efficient, system-wide Android dynamic analysis system TaintDroid was proposed to track the flow of sensitive data [21]. An improved version of TaintDroid named Droidbox was introduced in [19]. Portokalidis et al. proposed an alternative dynamic approach [49]. This approach performed the malware detection task on the remote servers in the cloud while the execution of Android software on the device was mirrored in virtual machine environments. In 2011, a crowdsourcing-based dynamic analysis approach was proposed to detect Android malware samples [15]. The detector was embedded in an integrated framework to collect different behavior traces of the candidate applications from a crowdsourcing system. The crowdsourcing strategy made it possible to capture real behaviors traces of a large number of applications. Shabtai et al. presented a dynamic host-based Android malware detection framework in [60].

Another dynamic analysis platform for Android named DroidScope, which could reconstruct Linux OS level and Java Dalvik level semantic information simultaneously and seamlessly was presented in [77]. To perform large-scale Android applications analysis, Rastogi et al. implemented an automatic dynamic analysis framework for Android named AppsPlayground [54]. AppsPlayground integrated various automatic detection or exploration techniques (e.e., a taint analysis tool [21], a kernel-level system call monitoring) to construct an effective dynamic analysis platform. Reina et al. implemented CopperDroid [55], a tool built on QEMU [12] to automatically analyze the out-of-box dynamic behaviors of Android malicious samples. In 2014, AirBag, a client-side approach that leveraged light-weight operating system level virtualization was presented to enhance the safety of the Android platform and to facilitate the defense capability against

Android malware [73]. Backes et al. introduced a genetic and extensible Android Security Framework (ASF) in [9].

In summary, the dynamic behavior analysis can easily discover the malicious behaviors that may miss out by static code analysis. Besides, it is effective in combating code encryption or obfuscation techniques [66,79]. However, the code coverage rate of dynamic behaviors analysis is lower than that of the static code analysis, thus it tends to miss some code sections that will be executed or triggered at certain time or scenarios (e.g., the advanced Android malware may hide or stop their malicious behaviors once they detect the virtual environment, the malicious activities may be triggered only at night). In addition, dynamic analysis techniques cost more computational resources.

2.3 Hybrid Analysis

The hybrid analysis technique combines both the static code analysis and the dynamic behaviors analysis. In other words, it not only analyses the source code of Android applications but also monitors the behaviors while the applications are actually executed [11].

In 2010, a system named AASandbox (Android Application Sandbox) was proposed to perform a hybrid analysis to automatically detect Android malicious samples [14]. In the static analysis part, AASandbox disassembled the *classes.dex* bytecode into the intermediate *Smali* code and then pre-checked the code that may imply malicious code segments. In the dynamic analysis part, the candidate Android applications were executed in the emulator for the behavior inspection. A comprehensive investigation for the detection of Android malware samples from both official and third-party stores using the hybrid analysis technique was presented in [86]. Firstly, a permission-based footprinting method was proposed to detect known-family malware samples. Then to detect the unknown malware samples, a heuristics-based filtering method was designed to identify the specific inherent behaviors of unknown malware families. In 2012, Zheng et al. addressed the challenging issue about how to activate the sensitive behaviors of Android applications in [84]. A hybrid analysis was proposed to uncover the UI-based trigger conditions through automated interactions. First of all, the static analysis was used to discover the expected activity switch paths by constructing Function Call and Activity Call Graphs. Then the dynamic analysis was performed to traverse each UI element and to investigate the UI interaction paths towards the sensitive APIs. Furthermore, the produced trigger conditions of the proposed approach could facilitate the existing dynamic tools, such as TaintDroid [21], to automatically identify the corresponding sensitive behaviors.

Another novel hybrid analysis system named Mobile-Sandbox was presented in [61]. Mobile-Sandbox employed specific techniques to track calls to native APIs (e.g., C/C++). In the static analysis, the *AndroidManifest.xml* and binary *classes.dex* were disassembled and analyzed to determine whether the candidate Android applications were performing potential suspicious permissions or intents. Then these applications were executed in the sandbox to log all behaviors including native API calls. EvoDroid [41] and A5 (Automated Analysis of

Adversarial Android Applications) [68] were two hybrid analysis systems similar to Mobile-Sandbox [61], which also utilized the static analysis to traverse all possible activity path to guide the further dynamic analysis. In [2], Afonso et al. conducted a large-scale hybrid analysis of Android applications in the wild to investigate how applications use the native code.

To address the code obfuscation issue, Rasthofer et al. presented HARVESTER to capture run time values from Android applications, even from those highly obfuscated advanced Android malware [53]. HARVEST could boost the recall of the existing analysis tools, such as the dynamic tool TaintDroid [21] and the static tool FlowDroid [7].

A generic Android input generator named IntelliDroid was proposed in [72]. IntelliDroid could be configured to generate inputs specific for a dynamic analysis system. Two techniques were employed to activate the targeted APIs with the injected inputs: identifying event chains and device-platform interface input injection. In addition, combining IntelliDroid with dynamic analysis tool TaintDroid [21] was able to provide better performance than FlowDroid [7].

In summary, the hybrid analysis technique exploits the advantages of static and dynamic analysis techniques. It not only captures the semantic structural information from the source code of Android applications but also tracks their running behaviors. Therefore, the hybrid analysis technique is able to adapt to code obfuscation while increasing the code coverage rate. However, hybrid analysis consumes expensive resources, and it requires a longer time to produce the analysis results [66,79]. Thus the usability of hybrid analysis is limited in a practical deployment.

3 Machine Learning Involved Android Malware Analysis

Given the soaring number of Android applications from both official and third-party stores, security experts or vendors have to inspect them in a short period to figure out their purposes or capabilities. Then the corresponding countermeasures will be provided based on the inspection results [61]. Thus it is necessary to accelerate the malware analysis process with little or even no human interventions. The Machine Learning techniques, which have been widely used in many cyber security areas [34,40,51,63,71,75,82], open the door for an alternative perspective to effectively and automatically identify or classify Android malicious samples. This section reviews the Android malware analysis methods using machine learning methods. Figure 2 presents the general framework of machine learning based malware analysis. The framework mainly consists of four steps: First of all, collecting the raw Android applications (including both benign and malicious samples) and setting up the ground truth (malicious/benign or specific family class). Second, performing feature engineering to extract informative features to characterize Android application samples. Third, training machine learning models for the following malware detection or classification. Fourth, predicting the candidate samples, evaluating the model and explaining the results. In the following, we will review the related works based on each step of the framework.

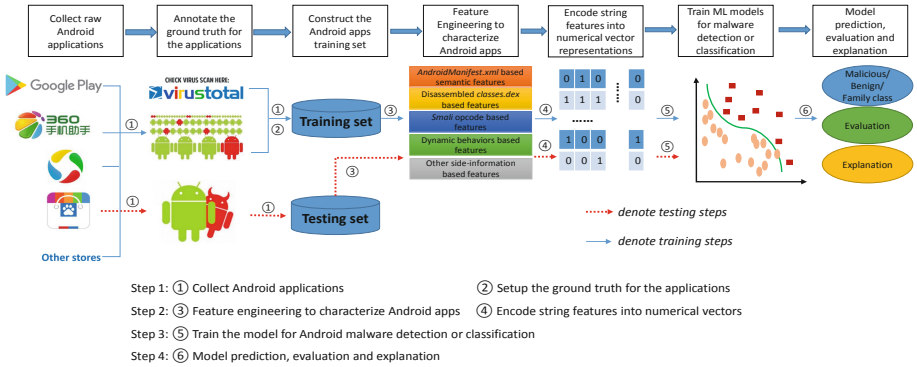

Fig. 2. The general Machine Learning framework for binary Android malware detection or multi-class family classification.

3.1 Raw Android Application Data Collection

Collect Raw Android Applications

The raw Android applications can be downloaded from various sources, e.g., the official Google Play store and the alternative third-party app stores (such as WanDouJia, AnZhi). For large-scale and up-to-date access to Android applications, it is necessary to implement specific crawlers for different Android application stores to automatically browse, retrieve, and download applications into repository [4,39].

However, there exist many challenges in crawling Android applications [4]. Firstly, different app stores have specific policies to limit or forbid the crawling of applications. For example, without a verified Google account, users cannot download any Android applications from Google Play. In addition, a valid account is allowed to download a limited number of applications in a given time from one IP address. Secondly, during the application downloading process, the crawlers have to be adapted to the updates of the stores. For example, if the application stores change the structure of the HTML pages, then a new analysis of the web pages is required to revise the crawling scripts. Thirdly, for a given application, it is difficult to download its previous versions. Most stores only provide the latest version of applications. Thus, it is hard to guarantee that all versions of the applications have been downloaded [39].

Annotate the Ground Truth of Applications

Reliable ground truth data is essential for evaluating the effectiveness of the malware analysis approaches. When the raw Android applications have been downloaded, the next step is to annotate the ground truth (identify the malicious samples or further assign a family class to them) to construct the training set. Most of the malware labeling works employed the state-of-the-art open-source

online scanning service VirusTotal[9] to annotate the labels for Android applications. VirusTotal incorporates more than 70 anti-virus tools and URL/domain blacklisting services to provide a comprehensive analysis report for each uploaded sample. Give a candidate Android application, the detailed ground truth annotation steps are as follows: First of all, determining whether the application is malicious or not using the majority voting strategy based on the results of different anti-virus tools in VirusTotal. Second, if the candidate application is malicious, we can further assign it a family class based on the returned analysis results of different anti-virus tools. However, there exists the inconsistent family naming issue from different anti-virus tools. Thus it is challenging to assign an accurate family class to the candidate malware sample.

Currently, there were two state-of-the-art works focusing on the Android malware family class annotation. The first piece of work was based on the dominant keyword algorithm [69]. First of all, the keywords from each of the detection reports of anti-virus tools were extracted. Then the generic keywords were filtered out. Finally, the rest keywords were counted to identify the dominant keyword, which was thus considered as the family name. The second piece of work was AVclass proposed in [59]. AVclass utilized new techniques to address three issues: normalization, removal of generic tokens, and alias detection. Thus AVclass was able to generate the most likely family names for a massive number of Android malware samples based on the detection reports of selected anti-virus tools.

3.2 Feature Engineering for Application Representation

The key building block in machine learning involved methods is feature engineering. Extracting the informative and robust features to represent Android applications is critical to the effectiveness and reliability of the models. In general, the common features used to characterize Android applications can be roughly divided into four categories: *AndroidManifest.xml* based semantic features. Disassembled *classes.dex* based semantic features. Intermediate *Smali* opcode based features. Fourth, the dynamic behaviors based features and other side-information based features.

AndroidManifest.xml-Based Features

Each Android application package contains the *AndroidManifest.xml* file. This file presents the essential information of the application, such as *Hardware components*, *Requested permissions*, *App components*, and *Filtered intents*. The stored information in this file can be parsed efficiently through static analysis [5]. Table 1 shows the detailed information of the features that can be extracted from *AndroidManifest.xml* to characterize Android samples.

Disassembled classes.dex-Based Features

The Android application package is usually implemented using Java programming language and then compiled into *classes.dex* bytecode for its execution

[9] https://www.virustotal.com/.

Table 1. The detailed information of the features that can be parsed from *AndroidManifest.xml*.

Feature subset	Detailed description of the feature subset
Requested permission	Android apps will request permissions for accessing critical resources during installation
App components	Android apps can declare many components, for instance, Service, Activity, Broadcastreceiver, ContentProvider
Filtered intents	Android apps use intent filters to appoints the operations it can perform and the data type it can manipulate
Hardware components	Apply specific hardware or a series of particular hardwares may imply potential security or privacy risks

in the Dalvik virtual machine. The *classes.dex* bytecode contains the comprehensive semantic knowledge about the critical API calls and data access within an application [5]. Besides, the *classes.dex* bytecode can be efficiently disassembled and parsed to represent Android applications. Table 2 shows the detailed descriptions of the low-level features that can be captured through disassembling *classes.dex*. Some high-level graph features, e.g., control flow graph [8,78], API dependency graph [83], code property graph [76], and inter-component call graph [25,28] can also be extracted from *classes.dex*.

Table 2. The detailed information of the low-level features that can be captured from *classes.dex*.

Feature subset	Detailed description of the feature subset
Suspicious API calls	The suspicious API calls represent the potential malicious actions of malware
Restricted API calls	The restricted API calls reveal the critical capability of Android applications
Used permissions	The restricted API calls will be used to decide and match the requested or indeed used permissions
Network addresses	The network addresses appeared in the source codes are related to potential botnet attacks or suspicious websites

Intermediate Smali Opcode-Based Features

Smali code is the intermediate but interpreted code between Java and Dalvik virtual machine. All the *Smali* codes follow a set of grammar specifications. The *classes.dex* can be disassembled into a set of *Smali* format files. Each *Smali* file represents a single class containing all the methods within the class and each method contains human-readable Dalvik instructions. Each instruction can be parsed into a single opcode and multiple operands [36]. To reduce the noise and improve efficiency, the common Dalvik instructions can be further categorized

into 7 core instruction sets while discarding the operands as shown in Table 3. Then n-grams features can be extracted from the opcode sequences of all the classes of an Android application.

Table 3. The descriptions of the 7 types of *Smali* opcode instruction sets.

Instruction type	The involved instructions
Move (M)	move, move/from16, move/16, move-wide, move-wide/from16, move-wide/16, move-object, move-object/from16, move-object/16, move-result, move-result-wide, move-result-object, move-exception
Return (R)	return-void, return, return-wide, return-object
Goto (G)	goto, goto/16, goto/32
If (I)	if-eq, if-ne, if-lt, if-ge, if-gt, if-le, if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez
Get (T)	aget, aget-wide, aget-object, aget-boolean, aget-byte, aget-char, aget-short, iget, iget-wide, iget-object, iget-boolean, iget-byte, iget-char, iget-short, sget, sget-wide, sget-object, sget-boolean, sget-byte, sget-char, sget-short
Put (P)	aput, aput-wide, aput-object, aput-boolean, aput-byte, aput-char, aput-short, iput, iput-wide, iput-object, iput-boolean, iput-byte, iput-char, iput-short, sput, sput-wide, sput-object, sput-boolean, sput-byte, sput-char, sput-short
Invoke (V)	invoke-virtual, invoke-super, invoke-direct, invoke-static, invoke-interface, invoke-virtual/range, invoke-super/range, invoke-direct/range, invoke-static/range, invoke-interface-range, invoke-direct-empty, invoke-virtual-quick, invoke-virtual-quick/range

Dynamic Behaviors-Based Features

The dynamic analysis tools can track abundant behaviors information of Android applications during actual execution. These behaviors information, e.g., file or network operations, information leaks can be efficiently parsed to represent Android applications. Table 4 lists the 10 common dynamic behavior feature set that can be used to characterize Android application samples [24].

Other Side-Information-Based Features

In addition to the features extracted directly from the static or dynamic analysis, other side-information-based features can also be parsed to characterize Android applications [20,56,87]. Zhu et al. proposed a method named FeatureSmith to automatically engineering features for malware detection by mining the security literature [87]. The natural language techniques were employed for mining Android documents (e.g. scientific or academic papers) and for representing and retrieving the semantic information about malware. Besides, the metadata, such

Table 4. The detailed introduction of 10 common dynamic behavior feature sets.

Dynamic behavior	Detailed description of the behavior
File operations	Scanning the file-system to retrieve sensitive data or creating external files to store the data
Network operations	Receiving bot commands from C& C servers or fetching malicious payloads from malicious websites
Cryptographic operations	Encrypting root exploits, targetd premium SMS number, critical methods, malicious payloads or URLs to evade detection
Information leaks	Collecting sensitive data (IMEI, account credentials, SMS, contact lists) and sending them to remote server
Dexclass load	Loading malicious payloads from app's assets, from another app or from remote system at running
Phone calls	Making phone calls stealthily without users' awareness
Sent SMS	Causing financial charges to infected devices by subscribing premium-rate services
Receiver actions	Malware usually exploits system events to trigger malicious payloads, while receivers are good indicators of system events
Service start	Malicious behaviors usually perform in background processes contained in Android's service components
System calls	System calls show how applications request services from operating system's kernel

as the profile information of Android applications or the profile of application developers can also be parsed to represent Android applications [45]. Furthermore, the software complexity metrics, e.g., the *Chidamber and Kemerer Metrics Suite* [17] and *McCabe's Cyclomatic Complexity* [43], can be employed to characterize Android application samples [13,50].

3.3 Model Training for Malware Detection or Classification

In this step, the machine learning models will be trained for Android malware detection or classification. Currently, both traditional machine learning models (e.g., Support Vector Machine [5], Random Forest [42], K-Nearest Neighbors [42]) and deep learning models (e.g., Deep Neural Networks [31,80,81], Convolutional Neural Networks [33]) have been applied to malware analysis. In addition, for a specific malware detection issue, some particular machine learning algorithms were also employed. For example, in [32], to aggregate different similarities between Android applications, multi-kernel learning [65] was applied to automatically learn the weights of different similarity perspectives.

3.4 Model Prediction, Evaluation, and Explanation

In this step, the trained machine learning model will be used to detect or classify the candidate Android applications in the wild. Generally, there is no ground truth data available for the Android samples in the wild. To evaluate the effectiveness of the proposed approach, it is common to divide part of the labeled Android applications as the testing set to validate the efficiency and efficacy of the model.

Since Android malware detection or family attribution are class-imbalanced classification problem, thus *Accuracy* along is far from enough to comprehensively evaluate the effectiveness of the models, more metrics, e.g., *Recall*, *Precision* or *F1-score* should be introduced. In practice, an effective machine learning based malware detection or classification approach should work with high accuracy as well as high efficiency. Generally, the efficiency refers to prediction time, because in most cases the training process can be finished offline. Therefore training time may not be a key challenge in the Android malware analysis while prediction time is really important especially the trained model is deployed on the mobile devices with limited computation resources.

Android malware samples constantly evolve over time. Thus it is important that the proposed approach is able to adapt to the evolution or population drift of malware (e.g., code obfuscation or encryption). *Adaptiveness* is a metrics used to explore whether malware detectors are able to learn fresh patterns while unlearning the obsolete patterns of malicious samples with time evolving [46].

In practice, an Android malware detection method must not only identify malicious samples but also offer explanations for the corresponding detection results [5]. The existing works provided different explanation granularity [5,16,27]. For example, in [5], the explanation consisted of a ranked list of features most indicative of malicious behavior and the corresponding weights reflecting their relative contribution to the detection results. In [27], the explanation results could locate the malicious components and the corresponding suspicious metadata (e.g., sensitive data leaked by the component).

4 Possible Future Research Directions

In this section, we briefly present the future research directions of Android malware detection. It is known that malware detection is a fundamental and indispensable topic in Cyber Security area. Researchers, as well as security vendors have invested a considerable amount of time and money to address this topic. And Machine learning techniques have been applied to Android malware detection for almost ten years. The future research directions will focus on fine-grained analysis, the details are as follows:

Firstly, as mentioned in the previous section, the source code of Android applications can be regarded as a special format natural language text, thus the Natural Language Processing (NLP) techniques can be employed to facilitate the detection performance. Besides, the NLP technique can be employed to better capture the semantic meanings within and between Android applications [48].

Therefore, the combination of Android malware detection and NLP technique will be a promising research direction [3].

Secondly, it has been shown that lots of detection methods or commercial tools are good at detecting specific type or family of malware. However, the detection performance is unsatisfactory when extending to other types of malware. Besides, the detection approaches are necessary to adapt to different versions of the same malware across various versions of the device OS. To address this issue, Transfer Learning may be a potential future direction [70].

Thirdly, concept drift in Android malware detection is a serious issue whereby models trained using older malware are not able to detect newer malware with confidence. Thus identifying such antiquated detection models accurately and timely is vital to the final performance. Traditional ML framework shown in Fig. 2 had to re-train frequently to adapt to the latest landscape of malware. To address this issue, online learning techniques can be introduced to fight against concept drifting by updating the model continuously and efficiently with the most recent malware examples [29,35,46].

Fourthly, inspired by the breakthroughs of Deep Learning in image classification, machine translation and natural language processing [18,30], Deep Learning has been introduced to malware detection and achieved satisfactory performance [44,80]. It can be expected that the latest Deep Learning models will continue to be a potential approach for malware detection.

Essentially, Android malware detection can be regarded as a class-imbalanced classification problem. The number of malware is far less than the number of benign applications. However, to the best of our knowledge, there are few works targeting the imbalance characteristic of Android malware detection issue. Thus, the cost-sensitive classification approaches, which has been shown effective in tackling class-imbalanced problems, may be a future research focus [64].

5 Conclusion

Android malware detection is a fundamental and systematic research topic in cyber security. It has been widely studied by both academic communities and security corporations. Meanwhile, machine Learning technique has also been applied to Android malware detection for nearly ten years. However, there also exist several challenges and difficulties in Android malware analysis area. In this survey, the research status of Android malware detection was presented. On the first, like other surveys, we reviewed the traditional software engineering based Android malware analysis techniques (static, dynamic and hybrid techniques). Our main focus is the Machine Learning framework for Android malware detection. We presented the detailed introduction of each part of the Machine Learning framework. Then, we gave the possible research directions about Android malware detection. In the end, we concluded the full survey.

References

1. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Zia, T., Zomaya, A., Varadharajan, V., Mao, M. (eds.) SecureComm 2013. LNICSSITE, vol. 127, pp. 86–103. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-04283-1_6
2. Afonso, V.M., et al.: Going native: using a large-scale analysis of Android apps to create a practical native-code sandboxing policy. In: NDSS. The Internet Society (2016)
3. Allamanis, M., Barr, E.T., Devanbu, P.T., Sutton, C.A.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. **51**(4), 81:1–81:37 (2018)
4. Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L.: Androzoo: collecting millions of Android apps for the research community. In: MSR, pp. 468–471. ACM (2016)
5. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of Android malware in your pocket. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, 23–26 February 2014 (2014)
6. Arshad, S., Shah, M.A., Khan, A., Ahmed, M.: Android malware detection & protection: a survey. Int. J. Adv. Comput. Sci. Appl. **7**(2), 463–475 (2016)
7. Arzt, S., et al.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, United Kingdom, 09–11 June 2014, pp. 259–269 (2014)
8. Atici, M.A., Sagiroglu, S., Dogru, I.A.: Android malware analysis approach based on control flow graphs and machine learning algorithms. In: 2016 4th International Symposium on Digital Forensic and Security (ISDFS), pp. 26–31. IEEE (2016)
9. Backes, M., Bugiel, S., Gerling, S., von Styp-Rekowsky, P.: Android security framework: extensible multi-layered access control on Android. In: Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, 8–12 December 2014, pp. 46–55 (2014)
10. Bartel, A., Klein, J., Traon, Y.L., Monperrus, M.: Dexpler: converting Android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, 14 June 2012, pp. 27–38 (2012)
11. Baskaran, B., Ralescu, A.: A study of Android malware detection techniques and machine learning. In: Proceedings of the 27th Modern Artificial Intelligence and Cognitive Science Conference 2016, Dayton, OH, USA, 22–23 April 2016, pp. 15–23 (2016)
12. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, Anaheim, CA, USA, 10–15 April 2005, pp. 41–46 (2005)
13. Beyer, D., Fararooy, A.: A simple and effective measure for complex low-level dependencies. In: ICPC, pp. 80–83. IEEE Computer Society (2010)
14. Bläsing, T., Batyuk, L., Schmidt, A., Çamtepe, S.A., Albayrak, S.: An Android application sandbox system for suspicious software detection. In: 5th International Conference on Malicious and Unwanted Software, MALWARE 2010, Nancy, France, 19–20 October 2010, pp. 55–62 (2010)
15. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, SPSM 2011, Co-Located with CCS 2011, Chicago, IL, USA, 17 October 2011, pp. 15–26 (2011)

16. Chen, K., et al.: Finding unknown malice in 10 seconds: mass vetting for new threats at the Google-play scale. In: USENIX Security Symposium, pp. 659–674. USENIX Association (2015)
17. Churcher, N.I., Shepperd, M.J.: Comments on "a metrics suite for object oriented design". IEEE Trans. Softw. Eng. **21**(3), 263–265 (1995)
18. Costa-jussà, M.R., Allauzen, A., Barrault, L., Cho, K., Schwenk, H.: Introduction to the special issue on deep learning approaches for machine translation. Comput. Speech Lang. **46**, 367–373 (2017)
19. Desnos, A., Lantz, P.: DroidBox: an Android application sandbox for dynamic analysis. Technical report, Lund University, Lund, Sweden (2011)
20. Dumitras, T.: Automatic feature engineering: learning to detect malware by mining the scientific literature (2017)
21. Enck, W., et al.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, Vancouver, BC, Canada, 4–6 October 2010, pp. 393–407 (2010)
22. Enck, W., Ongtang, M., McDaniel, P.D.: On lightweight mobile phone application certification. In: Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, 9–13 November 2009, pp. 235–245 (2009)
23. Enck, W., Ongtang, M., McDaniel, P.D.: Understanding Android security. IEEE Secur. Priv. **7**(1), 50–57 (2009)
24. Feng, P., Ma, J., Sun, C., Xu, X., Ma, Y.: A novel dynamic Android malware detection system with ensemble learning. IEEE Access **6**, 30996–31011 (2018)
25. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of Android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 576–587. ACM (2014)
26. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of Android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, 16–22 November 2014, pp. 576–587 (2014)
27. Feng, Y., Bastani, O., Martins, R., Dillig, I., Anand, S.: Automated synthesis of semantic malware signatures using maximum satisfiability. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, 26 February–1 March 2017 (2017)
28. Feng, Y., Wang, X., Dillig, I., Lin, C.: EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. In: OOPSLA, pp. 520–534. ACM (2015)
29. Gama, J., Zliobaite, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Comput. Surv. **46**(4), 44:1–44:37 (2014)
30. Hinton, G., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Sig. Process. Mag. **29**(6), 82–97 (2012)
31. Hou, S., Saas, A., Chen, L., Ye, Y.: Deep4maldroid: a deep learning framework for Android malware detection based on Linux kernel system call graphs. In: WI Workshops, pp. 104–111. IEEE Computer Society (2016)

32. Hou, S., Ye, Y., Song, Y., Abdulhayoglu, M.: HinDroid: an intelligent Android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, 13–17 August 2017, pp. 1507–1515 (2017)
33. Hsien-De Huang, T., Kao, H.-Y.: R2–D2: color-inspired convolutional neural network (CNN)-based Android malware detections. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 2633–2642. IEEE (2018)
34. Jiang, J.J., Wen, S., Yu, S., Xiang, Y., Zhou, W.: Identifying propagation sources in networks: state-of-the-art and comparative studies. IEEE Commun. Surv. Tutor. **19**(1), 465–481 (2017)
35. Jordaney, R., et al.: Transcend: detecting concept drift in malware classification models. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, 16–18 August 2017, pp. 625–642 (2017)
36. Kang, B., Yerima, S.Y., McLaughlin, K., Sezer, S.: N-opcode analysis for Android malware classification and categorization. In: 2016 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), pp. 1–7. IEEE (2016)
37. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Handbook of Satisfiability, pp. 613–631 (2009)
38. Li, L., et al.: Static analysis of Android apps: a systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017)
39. Li, L., et al.: Androzoo++: collecting millions of Android apps and their metadata for the research community. CoRR, abs/1709.05281 (2017)
40. Liu, L., de Vel, O.Y., Han, Q., Zhang, J., Xiang, Y.: Detecting and preventing cyber insider threats: a survey. IEEE Commun. Surv. Tutor. **20**(2), 1397–1417 (2018)
41. Mahmood, R., Mirzaei, N., Malek, S.: EvoDroid: segmented evolutionary testing of Android apps. In: SIGSOFT FSE, pp. 599–609. ACM (2014)
42. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G.J., Stringhini, G.: MaMaDroid: detecting Android malware by building Markov chains of behavioral models. In: 24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, 26 February–1 March 2017 (2017)
43. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976)
44. McLaughlin, N., et al.: Deep Android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, 22–24 March 2017, pp. 301–308 (2017)
45. Muñoz, A., Martín, I., Guzmán, A., Hernández, J.A.: Android malware detection from Google play meta-data: selection of important features. In: CNS, pp. 701–702. IEEE (2015)
46. Narayanan, A., Chandramohan, M., Chen, L., Liu, Y.: Context-aware, adaptive, and scalable Android malware detection through online learning. IEEE Trans. Emerg. Top. Comput. Intell. **1**(3), 157–175 (2017)
47. Narayanan, A., Chandramohan, M., Chen, L., Liu, Y.: A multi-view context-aware approach to Android malware detection and malicious code localization. Empirical Softw. Eng. **23**(3), 1222–1274 (2018)

48. Nguyen, T.D., Nguyen, A.T., Phan, H.D., Nguyen, T.N.: Exploring API embedding for API usages and applications. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 438–449 (2017)
49. Portokalidis, G., Homburg, P., Anagnostakis, K., Bos, H.: Paranoid Android: versatile protection for smartphones. In: Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6–10 December 2010, pp. 347–356 (2010)
50. Protsenko, M., Müller, T.: Android malware detection based on software complexity metrics. In: Eckert, C., Katsikas, S.K., Pernul, G. (eds.) TrustBus 2014. LNCS, vol. 8647, pp. 24–35. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09770-1_3
51. Qiu, J., Luo, W., Nepal, S., Zhang, J., Xiang, Y., Pan, L.: Keep calm and know where to focus: measuring and predicting the impact of Android malware. In: Gan, G., Li, B., Li, X., Wang, S. (eds.) ADMA 2018. LNCS (LNAI), vol. 11323, pp. 238–254. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05090-0_21
52. Qiu, J., Luo, W., Pan, L., Tai, Y., Zhang, J., Xiang, Y.: Predicting the impact of Android malicious samples via machine learning. IEEE Access **7**, 66304–66316 (2019)
53. Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in Android applications that feature anti-analysis techniques. In: NDSS. The Internet Society (2016)
54. Rastogi, V., Chen, Y., Enck, W.: AppsPlayground: automatic security analysis of smartphone applications. In: Third ACM Conference on Data and Application Security and Privacy, CODASPY 2013, San Antonio, TX, USA, 18–20 February 2013, pp. 209–220 (2013)
55. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In: EuroSec, April 2013
56. Sabottke, C., Suciu, O., Dumitras, T.: Vulnerability disclosure in the age of social media: exploiting Twitter for predicting real-world exploits. In: USENIX Security Symposium, pp. 1041–1056. USENIX Association (2015)
57. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: PUMA: permission usage to detect malware in Android. In: Herrero, Á., et al. (eds.) International Joint Conference CISIS 2012-ICEUTE 2012-SOCO 2012 Special Sessions. AISC, vol. 189, pp. 289–298. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33018-6_30
58. Sato, R., Chiba, D., Goto, S.: Detecting Android malware by analyzing manifest files. Proc. Asia-Pac. Adv. Netw. **36**(23–31), 17 (2013)
59. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
60. Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.: "Andromaly": a behavioral malware detection framework for Android devices. J. Intell. Inf. Syst. **38**(1), 161–190 (2012)
61. Spreitzenbarth, M., Freiling, F.C., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into Android applications. In: SAC, pp. 1808–1815. ACM (2013)

62. Suarez-Tangil, G., Stringhini, G.: Eight years of rider measurement in the Android malware ecosystem: evolution and lessons learned. CoRR, abs/1801.08115 (2018)
63. Sun, N., Zhang, J., Rimba, P., Gao, S., Zhang, L.Y., Xiang, Y.: Data-driven cybersecurity incident prediction: a survey. IEEE Commun. Surv. Tutor. **21**(2), 1744–1772 (2019)
64. Sun, Y., Kamel, M.S., Wong, A.K.C., Wang, Y.: Cost-sensitive boosting for classification of imbalanced data. Pattern Recogn. **40**(12), 3358–3378 (2007)
65. Sun, Z., Ampornpunt, N., Varma, M., Vishwanathan, S.: Multiple kernel learning and the SMO algorithm. In: Advances in Neural Information Processing Systems, pp. 2361–2369 (2010)
66. Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L.: The evolution of Android malware and Android analysis techniques. ACM Comput. Surv. **49**(4), 76:1–76:41 (2017)
67. Tang, W., Jin, G., He, J., Jiang, X.: Extending Android security enforcement with a security distance model. In: 2011 International Conference on Internet Technology and Applications, pp. 1–4. IEEE (2011)
68. Vidas, T., Tan, J., Nahata, J., Tan, C.L., Christin, N., Tague, P.: A5: automated analysis of adversarial Android applications. In: SPSM@CCS, pp. 39–50. ACM (2014)
69. Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current Android malware. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 252–276. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_12
70. Weiss, K.R., Khoshgoftaar, T.M., Wang, D.: A survey of transfer learning. J. Big Data **3**, 9 (2016)
71. Wen, S., Haghighi, M.S., Chen, C., Xiang, Y., Zhou, W., Jia, W.: A sword with two edges: propagation studies on both positive and negative information in online social networks. IEEE Trans. Comput. **64**(3), 640–653 (2015)
72. Wong, M.Y., Lie, D.: IntelliDroid: a targeted input generator for the dynamic analysis of Android malware. In: NDSS. The Internet Society (2016)
73. Wu, C., Zhou, Y., Patel, K., Liang, Z., Jiang, X.: AirBag: boosting smartphone resistance to malware infection. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, 23–26 February 2014 (2014)
74. Wu, D., Mao, C., Wei, T., Lee, H., Wu, K.: DroidMat: Android malware detection through manifest and API calls tracing. In: Seventh Asia Joint Conference on Information Security, AsiaJCIS 2012, Kaohsiung, Taiwan, 9–10 August 2012, pp. 62–69 (2012)
75. Wu, T., Wen, S., Xiang, Y., Zhou, W.: Twitter spam detection: survey of new approaches and comparative study. Comput. Secur. **76**, 265–284 (2018)
76. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: IEEE Symposium on Security and Privacy, pp. 590–604. IEEE Computer Society (2014)
77. Yan, L., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012, pp. 569–584 (2012)
78. Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.: DroidMiner: automated mining and characterization of fine-grained malicious behaviors in Android applications. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8712, pp. 163–182. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9_10

79. Ye, Y., Li, T., Adjeroh, D.A., Iyengar, S.S.: A survey on malware detection using data mining techniques. ACM Comput. Surv. **50**(3), 41:1–41:40 (2017)
80. Yuan, Z., Lu, Y., Wang, Z., Xue, Y.: Droid-sec: deep learning in Android malware detection. In: ACM SIGCOMM 2014 Conference, SIGCOMM 2014, Chicago, IL, USA, 17–22 August 2014, pp. 371–372 (2014)
81. Yuan, Z., Lu, Y., Xue, Y.: Droiddetector: Android malware characterization and detection using deep learning. Tsinghua Sci. Technol. **21**(1), 114–123 (2016)
82. Zhang, J., Xiang, Y., Wang, Y., Zhou, W., Xiang, Y., Guan, Y.: Network traffic classification using correlation information. IEEE Trans. Parallel Distrib. Syst. **24**(1), 104–117 (2013)
83. Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014, pp. 1105–1116 (2014)
84. Zheng, C., et al.: SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications. In: SPSM@CCS, pp. 93–104. ACM (2012)
85. Zheng, M., Sun, M., Lui, J.C.S.: Droid analytics: a signature based analytic system to collect, extract, analyze and associate Android malware. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013/11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2013/12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, 16–18 July 2013, pp. 163–171 (2013)
86. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative Android markets. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, 5–8 February 2012 (2012)
87. Zhu, Z., Dumitras, T.: FeatureSmith: automatically engineering features for malware detection by mining the security literature. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 767–778 (2016)