Erzincan Üniversitesi
Fen Bilimleri Enstitüsü Dergisi
2021, 14(1),331-356
ISSN: 1307-9085, e-ISSN: 2149-4584
Araştırma Makalesi

Erzincan University
Journal of Science and Technology
2021, 14(1), 331-356
DOI: 10.18185/erzifbed.806683
Research Article

Hybroid: A Novel Hybrid Android Malware Detection Framework

Abdullah Talha Kabakus[1*]

[1]Department of Computer Engineering, Faculty of Engineering, Duzce University

Geliş / Received: 06/10/2020, Kabul / Accepted: 24/02/2021

Abstract

Android, the most widely-used mobile operating system, attracts the attention of malware developers as well as benign users. Despite the serious proactive actions taken by Android, the Android malware is still widespread as a result of the increasing sophistication and the diversity of malware. Android malware detection systems are generally classified into two: (1) Static analysis, and (2) dynamic analysis. In this study, a novel Android malware detection framework, namely, *Hybroid*, was proposed which combines both the static and dynamic analysis techniques to benefit from the advantages of both of these techniques. An up-to-date version of Android, namely, *Android Oreo*, was specifically employed in order to handle the problem from an up-to-date perspective as the recent versions of Android provide new security mechanisms, which are discussed with this study. *Hybroid* was evaluated on a large dataset that consists of 10,658 applications, and the accuracy of *Hybroid* was calculated as high as 99.5% when it was utilized with the *J48* classification algorithm which outperforms the state-of-the-art studies. The key findings in consequence of the experimental result are discussed in order to shed light on Android malware detection.

Keywords: Android malware detection, mobile malware, mobile security, static analysis, dynamic analysis, Android

Hybroid: Benzersiz Hibrit Bir Android Kötücül Yazılım Tespit Uygulama Çatısı

Öz

Dünyanın en çok kullanılan mobil işletim sistemi olan Android, zararsız kullanıcılar gibi kötücül yazılım geliştiricilerin de ilgisini çekmektedir. Android tarafından ön alıcı ciddi eylemler alınmasına rağmen Android kötücül yazılım artan kötücül yazılım çeşitliliği ve karmaşıklığı sebebiyle hala yaygındır. Android kötücül yazılım sistemleri genellikle ikiye ayrılır: (1) Statik analiz ve (2) dinamik analiz. Bu çalışmada statik ve dinamik analizi birleştirerek her ikisinin de avantajlarından faydalanan *Hybroid* ismiyle benzersiz bir hibrit Android kötücül yazılım tespit uygulama çatısı sunulmuştur. Çalışmada tartışıldığı üzere Android'in yeni sürümlerinde sunulan yeni güvenlik mekanizmalarıyla birlikte problemi güncel bir bakış açısıyla ele almak için Android'in güncel bir sürümü, *Android Oreo*, kullanılmıştır. *Hybroid* 10.658 uygulamadan oluşan geniş bir verisetinde test edilmiş ve *Hybroid*'in doğruluğu *J48* sınıflandırma algoritması kullanıldığında en gelişkin uygulamaları geride bırakarak %99.5 kadar yüksek çıkmıştır. Deneysel sonuçlar neticesinde elde edilen en önemli bulgular Android kötücül yazılım tespitine ışık tutmak amacıyla tartışılmıştır.

Anahtar Kelimeler: Android kötücül yazılım tespiti, mobil kötücül yazılım, mobil güvenlik, statik analiz, dinamik analiz, Android.

1. Introduction

The number of global smartphone users is growing rapidly thanks to the easiness and accessibility provided by them. The *GSM Association* predicts that the number of unique mobile subscribers will reach 5.9 billion by 2025 which is equivalent to 71% of the world's population ("The Mobile Economy 2018," 2018). Android, Google's mobile operating system, has dominated the global smartphone market. According to a recent report by *IDC*, Android's global smartphone market share has reached 86.8% in the third quarter of 2018 ("IDC - Smartphone Market Share," 2019). During the annual developer conference, *Google I/O 2017*, Google has announced that there are 2 billion monthly active Android devices globally (Burke, 2017; Popper, 2017). The main reasons behind this popularity can be listed as follows: (1) Being an open-source distribution which has made it a favorite for both consumers and developers since that makes the operating system to be customized and extended by companies for free such as *OxygenOS*[1], (2) being developed with the help of a consortium of 84 technology firms namely *Open Handset Alliance*[2], (3) building on the Linux kernel which is developed with the contributions of almost 10,000 developers from a large number of companies, and is a well-designed and stable software as the result of 37 years of experience, (4) the high level of sophistication (Aresu, Ariu, Ahmadi, Maiorca, & Giacinto, 2015; Karbab, Debbabi, Derhab, & Mouheb, 2018; A. Kumar, Kuppusamy, & Aghila, 2018; "Who Writes Linux? Almost 10,000 Developers," 2013; Yang, Wang, Ling, Liu, & Ni, 2017). This popularity has also attracted the attention of malicious software (malware) developers as *Sophos* reports that the number of malware has risen to nearly 3.5 million in 2017 which was under 2.5 million in 2015 ("SophosLabs 2018 Malware Forecast," 2018). The official application market of Android, *Play Store*[3], contains malware even though it is regularly scanned for malware through an always-on service provided by Google named *Google Play Protect*[4] and the found ones are immediately removed from the store ("Android – Google Play Protect," 2021; Cunningham, 2017; Villas-Boas, 2018). But many devices are affected until they are removed from the store. For example, according to the reports, the Android malware '*Judy*' is thought to be reached 36.5 million devices through the *Play Store* (Morris, 2017). Similarly, '*FalseGuide*', which is an Android malware that hides its malicious actions in over forty fake companion guide applications for popular mobile games such as *Pokemon Go* and *FIFA Mobile* has affected about 2 million devices (M. Kumar, 2017). Most recently, *ESET* has reported an Android trojan that steals money from *PayPal* accounts even bypassing *PayPal*'s two-factor authentication (Stefanko, 2018). When the literature is investigated, the aims of Android malware can be listed as follows: (1) Privilege escalation, (2) turning the devices into bots that are ready to be controlled remotely, (3) causing financial charges (i.e. sending messages to premium numbers, etc.), and (4) collecting sensitive information (i.e. the user's bank detail, etc.) from devices (Arshad, Ahmed, Shah, & Khan, 2016; Fan, Sang, Zhang, Sun, & Liu, 2017; Grace, Zhou, Zhang, Zou, & Jiang, 2012; Kang, Jang,

[1] https://www.oneplus.com/3t/oxygenos
[2] https://www.openhandsetalliance.com
[3] https://play.google.com/store
[4] *Google Play Protect* is the built-in, real-time malware protection tool for Android that uses Google's machine learning algorithms to improve its malware knowledge-base and malware detection accuracy by scanning over 50 billion applications from over 2 billion devices.

Mohaisen, & Kim, 2015; King, Lampinen, & Smolen, 2011; Peng et al., 2012; Rastogi, Chen, & Enck, 2013; Shabtai et al., 2014; Stefanko, 2018; Suarez-Tangil, Tapiador, Peris-Lopez, & Ribagorda, 2014; Y. Wang, Zheng, Sun, & Mukkamala, 2013; X. Wei, Gomez, Neamtiu, & Faloutsos, 2012; Xue et al., 2017; Yang et al., 2017; Yerima, Sezer, McWilliams, & Muttik, 2013; Yu, Huang, & Yian, 2016; Zhou & Jiang, 2012).

The security mechanism of Android solely depends on the permissions that are granted by the end-user during the application installation time (Bläsing, Batyuk, Schmidt, Camtepe, & Albayrak, 2010; Di Cerbo, Girardello, Michahelles, & Voronkova, 2011; Felt, Chin, Hanna, Song, & Wagner, 2011; Grace, Zhou, Wang, & Jiang, 2012; Mahmood et al., 2012). Prior to Android 6.0 (*a.k.a.* Marshmallow), Android shows a prompt window such as the one presented in Fig. 1 that lists the dangerous permissions that the application, which is going to be installed, demands, and the end-user has no choices but to grant these permissions to the application in order to complete the installation.

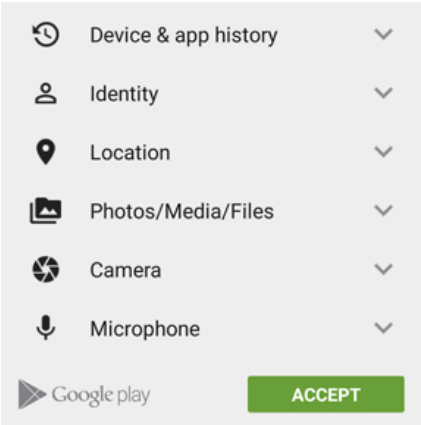

Figure 1. A sample prompt window displayed by Android, prior to Android 6.0, that lists the dangerous permissions demanded by the application during the installation.

By the release of Android 6.0, the end-user is not prompted during the installation regarding the dangerous permissions that the application demands. Instead of that, a prompt window, such as the one presented in Fig. 2, is being displayed to the end-user when the application needs the access grant of dangerous permissions during the runtime. This mechanism is also known as *runtime permissions* which lets the end-user revoke the grant access after the installation phase. In addition to this ability, the user may select the "*Deny & don't ask again*" option, which is displayed to users when the same permission is demanded by the application after denying, in order to always deny the dangerous action demanded by the application. Despite malware in the Android ecosystem is very common and risky, the researchers report that most of the end-users are unaware of the exact meanings and potential risks of granting these permissions, and they simply grant these permissions (Backes et al., 2012; Enck, Ongtang, & McDaniel, 2009; Gibler, Crussell, Erickson, & Chen, 2012; A.T. Kabakus & Dogru, 2018; Kelley et al., 2012; King et al., 2011; Mylonas, Kastania, & Gritzalis, 2013; Singh, Tiwari, & Singh, 2016; Yang et al., 2017). The applications are able to reach sensitive contents (i.e. messages, contacts, location, etc.) and use the hardware (i.e. camera, storage, sensors, etc.) of the smartphone if the related permission, which is declared in the application manifest file (namely *AndroidManifest.xml*), is granted by the end-user.

Figure 2. A sample prompt window displayed by the Android 6.0+ that asks the end-user's grant for a dangerous permission, namely, making and managing phone calls, during the runtime.

Android malware detection approaches in the literature are generally divided into two: (1) Static analysis which focuses on analyzing the application through its source files which are obtained thanks to the reverse engineering techniques without executing the application (Alzaylaee, Yerima, & Sezer, 2017; Enck et al., 2009; Fuchs, Chaudhuri, & Foster, 2009; Grace, Zhou, Zhang, et al., 2012; Zhou & Jiang, 2012), and (2) dynamic analysis which executes the application in an isolated environment (i.e. a sandbox, a virtual machine, etc.) to track the behavior (i.e. memory usage, network access, dynamic taint, etc.) and the effects of the application on that isolated environment (Alzaylaee et al., 2017; Chandramohan & Tan, 2012; Liang & Du, 2014; Singh et al., 2016; Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014). The advantages of static analysis are (1) it is fast compared to dynamic analysis as applications are not actually being executed in an isolated environment, (2) it provide a better code coverage as it evaluates all sources of an application, and (3) according to the related work, they are very effective in terms of malware detection. The main disadvantage of static analysis is that it fails to against the malware that protects its code through advanced obfustication techniques (e.g., code obfuscation) (Massarelli et al., 2017; Moser, Kruegel, & Kirda, 2007; Rhode, Burnap, & Jones, 2018; C. Wang, Li, Mo, Yang, & Zhao, 2017), dynamic loading techniques (e.g., reflection) (C. Wang et al., 2017; Zheng, Sun, & Lui, 2014), and encryption algorithms (Arshad et al., 2016; Bae & Shin, 2017; Tam, Feizollah, Anuar, Salleh, & Cavallaro, 2017; Tong & Yan, 2017; C. Wang et al., 2017). The main advantage of dynamic analysis is that it is able to come through advanced obfuscation and dynamic loading techniques as a result of examining applications during execution (Gadhiya, Bhavsar, & Student, 2013; Zheng et al., 2014). The main disadvantages of dynamic analysis are (1) it may miss some of the code sections that are not executed during the analysis (Arshad et al., 2016), and (2) it takes a relatively long time to analyze applications compared to static analysis. To this end, the main objectives of the proposed study, namely, *Hybroid*, are (1) benefitting from the advantages of both analysis techniques by proposing a novel hybrid approach that combines both of these techniques, and (2) extracting a novel feature set that can be used to efficiently detect Android malware. The rest of the paper is structured as follows: Section 2 presents the related work. Section 3 describes the material and method used within the proposed approach. Section 4 discusses the key findings and experimental results. Finally, Section 5 concludes the paper with future directions.

2. Related Work

In this section, the related work is briefly reviewed through the analysis technique it is based on, namely, (1) static analysis, and (2) dynamic analysis.

Static analysis

Stowaway (Felt et al., 2011) traces the API calls and aims to map them with the permissions in order to detect over-privileged applications. *Sato et al.* (Sato, Chiba, & Goto, 2013) extracts each application's malignancy score through the information extracted from the application manifest file, then calculated a malignancy score with threshold values. If the application's calculated malignancy score exceeds the threshold values, then the application is classified as malware. *PUMA* (Sanz et al., 2012) extracts the permissions and features defined in the application manifest file, then analyzes an application through its extracted information. Then it classifies applications thanks to the utilized machine learning algorithms which are the algorithms that extract information from raw data and represent it in some models (Kayikci, 2018). *Tang et al.* (Tang, Jin, He, & Jiang, 2011) use the security distance model to measure the dangerous level of an application due to the combinations of requested permissions. *AndroSimilar* (Faruki, Ganmoor, Laxmi, Gaur, & Bharmal, 2013) creates a variable-length signature to compare it with the signature database. It classifies an application as benign or malicious on the basis of the calculated similarity percentage. The biggest disadvantage of this approach when it is compared to the proposed one is that it can only detect known malware variants. *DroidAnalytics* (Zheng, Sun, & Lui, 2013) creates three-level signatures for each application on the basis of API calls and performs the op-code level analysis. Then it correlates the analysis with the existing malware in the database through the similarity score based on the class level signature. *Kirin* (Enck et al., 2009) compares the security configuration of an application with the security rules to mitigate malware at installation time by modifying the Android Application Installer. The proposed approach is specifically designed not to alter the core parts of the operating system itself (aka *rooting*). *SCanDroid* (Fuchs et al., 2009) analyzes an application's data flow in order to classify it as benign or malicious. *APK Auditor* (Abdullah Talha Kabakus, Dogru, & Cetin, 2015) is a permission-based static analysis system that extracts an application's permissions from the application's manifest file. Then it calculates a malware score through the extracted permissions by calculating a malware score for each permission which is calculated on the basis of how often the permission is demanded by the malicious or benign applications. If the application's calculated malware score exceeds the threshold score which is determined by utilizing the *Logistic Regression* algorithm, then the application is classified as malicious. *Sayfullina et al.* (Sayfullina et al., 2015) present an approach based on the extraction of three resources from the *apk* file namely (1) application manifest file, (2) *classes.dex* file which contains the compiled source code in *dex* (Dalvik executable) format which is a proprietary format for Java bytecode that is designed to be more compact and memory-efficient than regular Java class files (Shabtai, Kanonov, Elovici, Glezer, & Weiss, 2012), and (3) *resources.arsc* file which contains the compiled resources. Then they propose and utilize *Normalized Bernoulli Naïve Bayes* classifier in order to classify the applications based on these static resources. *Bao et al.* (Bao, Lo, Xia, & Li, 2017) propose two static analysis approaches which are (1) the approach that utilizes a collaborative filtering technique which is inspired from the intuition that applications that provide similar features usually demand similar permissions, and (2) the approach recommends permissions based on text mining technique that utilizes *Naïve Bayes Multinomial* classification algorithm to build

a prediction model by analyzing the description of applications which are available on the *Play Store*. The limitations of this approach can be listed as follows: (1) Not all applications do have descriptions (i.e. the applications in the datasets), (2) they detect API usages through the import statements but an import statement does not guarantee that the related API is used which is known as "unused import" in the context of Java programming language, and (3) they only consider the classes, which are available in Android SDK and Java standard libraries but the developers may define their own classes, and those classes may extend the classes in Android SDK or Java standard library thanks to the inheritance mechanism provided by Java programming language. *DroidDet* (Zhu et al., 2017) utilizes the *Rotation Forest* algorithm based on the static analysis features such as permissions, system events, and the rate of sensitive API calls to classify applications. *Significant Permission IDentification* (J. Li et al., 2018) detects Android malware based on permission usage. But unlike the related work, they do not extract and analyze all the permissions defined in the application manifest file. Instead of that, they propose three levels of pruning by mining permission data to identify the most significant permissions that can be effective in classifying Android applications as benign or malicious. *MalDozer* (Karbab et al., 2018) uses raw sequences of API calls based on neural networks and utilizes an automatic feature extraction technique during the training using *method embedding* where the input is the raw sequence of API calls that are extracted from DEX assembly. *Kim et al.* (Kim, Kang, Rho, Sezer, & Im, 2019) propose the first study of the multimodal deep learning to be used for Android malware detection. The proposed approach uses seven features, namely, (1) String feature, (2) method opcode feature, (3) method API feature, (4) shared library function opcode feature, (5) permission feature, (6) component feature, and (7) environmental feature.

Dynamic analysis

Androidetect (L. Wei et al., 2017) uses the process injection technique, Hook method, and inter-procedural communication that constructs the eigenvectors by extracting the characteristics of the Android application. *CrowDroid* (Burguera, Zurutuza, & Nadjm-Tehrani, 2011) performs system call tracing thanks to the *strace*[5] tool. Then *CrowDroid* client application creates a log file regarding the system calls and sends it to a remote server for deep analysis. *K-means* clustering algorithm is utilized through the constructed feature vectors. The limitation of the *CrowDroid* is the need for the installation of a *CrowDroid* client application to perform malware detection. Also, if a benign application uses more system calls, then it may be classified as malicious by *CrowDroid*. *MADAM* (Dini, Martinelli, Saracino, & Sgandurra, 2012) combines features that both the kernel-level and application-level and perform malware analysis. The drawback of the *MADAM* is that it performs whole analysis on the device which is not applicable since mobile devices generally have limited computation (e.g., CPU) and storage capabilities (e.g., memory, disk, battery) (Dini et al., 2012; Liu, Yan, Zhang, & Chen, 2009; Tam et al., 2017). *MimeoDroid* (Faruki, Zemmari, Gaur, Laxmi, & Conti, 2016) extracts features such as CPU and memory usages, Binder IPC transfers, network interaction, battery charging status, and permissions defined in the application manifest file in order to detect malicious behavior by utilizing tree-based machine learning classifiers. *Andromaly*

[5] https://strace.io

(Shabtai et al., 2012) is an on-device anomaly detector that analysis features such as SMS, voice call, data sent/received, and battery usage to detect anomalous malware behavior. Then *Andromaly* utilizes machine learning algorithms to classify applications as malicious or benign. *TaintDroid* (Enck et al., 2010) is a system-wide dynamic taint tracking and analysis system that simultaneously tracks multiple sources of sensitive data. *AppsPlayground* (Rastogi et al., 2013) proposes a real device-based sandbox emulation that utilizes malware detection techniques such as taint tracing for information leakage detection (which is based on *TaintDroid*), sensitive API monitoring, and kernel-level monitoring for the detection of root exploits. *Paranoid Android* (Portokalidis, Homburg, Anagnostakis, & Bos, 2010) transfers the execution trace which is recorded by a tracer located in the mobile device to a remote server which is responsible for replaying the execution trace within the replica of the mobile device. *Paranoid Android* has a similar limitation to *CrowDroid* which is the need for the installation of its tracer on the mobile device. *AntiMalDroid* (Zhao, Ge, Zhang, & Yuan, 2011) monitors the behavior of applications and their characteristics, then it categorizes these characteristics as malicious or benign. Then the learning module generates signatures through the extracted characteristics. If the application's signature matches an existing benign application's signature in the database, then the application is classified as benign. If the application's signature matches a malicious application's signature in the database, this time the application is classified as malicious.

3. Material and Method

In this section, the material and method used to develop the proposed novel hybrid Android malware detection framework, namely, *Hybroid*, are described.

Static analysis

An Android application is packaged as an *apk* (Android Application Package) file which is basically an archive file that contains all the sources and resources of the application. A Python script was implemented as a part of the proposed framework that is responsible for (1) extracting features from the application manifest file, and (2) storing the extracted features in the database. The input of the Python script is the *apk* file of the analyzed application. A tool, namely, *apktool*, was used to extract the content of the *apk* files. This tool was executed through the implemented Python script using the *subprocess*[6] module of Python. Then from the extracted archive, the application manifest file was parsed in order to retrieve the information related to the application. The Android permissions are categorized into three as follows ("Permissions Overview | Android Developers," 2019):

- *Normal permissions* are those that contain very little risk to the user's privacy or the operation of other applications. Hence, this type of permission is automatically granted by the system at install time without prompting the user to grant them.
- *Dangerous permissions* cover areas where the application wants to access sensitive data or could potentially affect the user's stored data or the operation of other applications through harmful API calls (Felt et al., 2011).
- *Signature permissions* are granted by the system when the app that attempts

[6] https://docs.python.org/3/library/subprocess.html

to use permission that is signed by the same certificate as the app that defines the permission.

Alongside these built-in permissions types, which are defined in the Android SDK, the developers may define their own permissions. This type of permissions is called "*custom permissions*". Alongside the permissions, an application's activities, services, features, and receivers are declared in the application file. This information was also extracted and stored in the database. Any component that is not declared in the application manifest file, cannot be used in the application. Otherwise, the application simply crashes. Obviously, this rule is valid for the permissions, as well. The API calls are mapped with the built-in permissions and the developer should declare the related permission in order to use the API call(s). In addition to the information defined in the application's manifest file, the total number of the lines of code in the application source code files was calculated after (1) decompiling the Dalvik executable file (*classes.dex*) into a Java archive file (*jar*) by using the *dex2jar* ("Pxb1988/Dex2jar: Tools to Work with Android .Dex and Java .Class Files," 2018) tool, and (2) extracting the *jar* file into the readable Java source code files by using the *jd-cmd* (Cacek, 2018) tool. All of these operations were handled automatically thanks to the implemented Python scripts. The whole static features of *Hybroid* are listed in Table 1 with their descriptions.

Table 1. The static analysis features of *Hybroid*.

Feature	Description
numOfDangerousPermissions	The number of dangerous permissions defined in the application manifest file
numOfCustomPermissions	The number of custom permissions defined in the application manifest file
numOfOtherPermissions	The number of other permissions defined in the application manifest file
numOfFeatures	The number of features defined in the application manifest file
numOfReceivers	The number of broadcast receivers defined in the application manifest file
numOfServices	The number of services defined in the application manifest file
numOfActivities	The number of activities defined in the application manifest file
loc	The total number of lines of code in the application's source code files

Dynamic analysis

The built-in emulator of Android, which comes with the Android SDK, was used to execute the application in order to monitor its runtime behavior. The utilized emulator's hardware and software specifications are listed in Table 2.

Table 2. The utilized emulator's hardware and software specifications.

Specification	Value
Operating System	Android Oreo 8.1 x86
API Level	27
RAM	1,536 MB
VM Heap	128 MB
Network Speed	Full
Network Latency	None

The *Monkey* tool, which is distributed with the Android SDK, is designed to provide a random testing tool that generates random events to interact with the applications to developers (Afonso, de Amorim, Grégio, Junquera, & de Geus, 2015; Alzaylaee et al., 2017; Canfora, Medvet, Mercaldo, & Visaggio, 2015; Spreitzenbarth, Freiling, Echtler, Schreck, & Hoffmann, 2013). In order to manage the emulator programmatically and monitor the application's behavior and traces, a Python script was implemented. Within the implemented Python script, a Python library namely *AndroidViewClient* (Milano, 2018) was utilized in order to interact with the emulator. The implemented Python script accepts the *apk* file as the input, and responsible for (1) installing the application, (2) detecting the launchable activity of the application through the *aapt* (Android Asset Packaging Tool), which is a tool provided by the Android SDK, (3) executing the application through the detected launchable activity, (4) waiting for the application to be launched for 5 seconds, and (5) generating 500 random events. The event generation was configured as follows: (1) Wait 1 second between each event, (2) the percentage of the system key events is set to 0 in order to prevent being pressed system keys such as *HOME*, and *BACK*, and (3) the timeouts and crashes were ignored. As a natural consequence of being built on the Linux Kernel, Android assigns a unique process id (*PID*) for each running process (application), and the *PID* of the executed application was detected in order to retrieve the detail regarding the application's CPU, memory, and network usages from the files in the */proc* filesystem. As a total, 30 features, which are listed in Table 3, were extracted from the files in the */proc* filesystem. When the execution of an application finishes, the obtained result of the dynamic analysis is stored in the database. Each time the emulator is started for a new analysis, it is restored to its initial status, which is a clean install of the Android operating system by wiping all the data created during the previous application execution in order to prevent any side effects. The average duration for an application to be analyzed is calculated as 70 seconds. The analysis was carried out on a computer with Intel Core i7-7700HQ CPU of 2.80 GHz clock speed and 32 GB of RAM of 2.40 GHz speed.

Table 3. The analyzed files in the */proc* filesystem and their descriptions.

File	Description
/proc/stat	Stores information about process status
/proc/[PID]/stat	Stores detailed information about the process
/proc/net/dev	Stores information related to network usage
/proc/[PID]/statm	Stores detailed information about process memory status information

The content of a sample */proc/stat* file is presented in Fig. 3 as the content of the samples of the other analyzed files is presented in the Appendix. The features extracted from each analyzed file in the */proc* filesystem are listed in the following tables.

cpu 10786 12539 13662 704375 10081 3 833 0 0 0
cpu0 2390 2840 3145 175753 1974 3 810 0 0 0
cpu1 2437 3249 3047 176489 2817 0 8 0 0 0
cpu2 2587 3283 3735 176414 2838 0 9 0 0 0
cpu3 3372 3167 3735 175719 2452 0 6 0 0 0
intr 1132090 20 0 0 0 2 0 0 0 1 0 46252 13272 0 0 0 0 2 0 65688 1 0 6539 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 2292635
btime 1546290061
processes 7423
procs_running 1
procs_blocked 0
softirq 595539 23106 112270 625 14135 0 0 235143 92951 524 116785
```

Figure 3. The content of a sample */proc/stat* file.

Table 4. The features that were extracted from the */proc/stat* file with their descriptions according to the Linux documentation[7] (Bowden, Bauer, Nerin, Feng, & Seibold, 2018).

Feature	**Description**
userCpu	Normal processes executing in user mode
niceCpu	Niced processes executing in user mode
systemCpu	Processes executing in kernel mode
idleCpu	Twiddling thumbs
irqCpu	Servicing interrupts
softirqCpu	Servicing softirqs
stealCpu	Involuntary wait
guestCpu	Running a normal guest
guestNiceCpu	Running a niced guest

[7] The *iowait* feature is intentionally excluded since reading it from */proc/stat* is not reliable according to the Linux documentation.

Table 5. The features that were extracted from the */proc/net/dev* file with their descriptions.

Feature	**Description**
rxBytes	Received packets in bytes
rxPackets	Number of received packets
txBytes	Transmitted packets in bytes
txPackets	Number of transmitted packets

Table 6. The features that were extracted from the */proc/[PID]/stat* file with their descriptions according to the Linux documentation (Bowden et al., 2018).

Feature	**Description**
cmajFlt	Number of major faults with child's
cminFlt	Number of minor faults with child's
cstime	Kernel mode jiffies with child's
cutime	User mode jiffies with child's
majFlt	Number of major faults
minFlt	Number of minor faults
numThreads	Number of threads
rss	Resident set memory size
vss	Virtual memory size
taskCpu	Which CPU the task is scheduled on
utime	User mode jiffies
stime	Kernel mode jiffies

Table 7. The features that were extracted from the */proc/[PID]/statm* file with their descriptions according to the Linux documentation (Bowden et al., 2018).

Feature	**Description**
vmSize	Total program size (pages)
residentSize	Size of memory portions (pages)
numPageShared	Number of pages that are shared
numPageText	Number of pages that are 'code'

numPageData	Number of pages of data/stack

Sample screenshots that were captured during the executions of the malicious applications are presented in Fig. 4. Android malware tends to hide its malicious actions (B. Li, Zhang, Li, Yang, & Gu, 2018). As can be seen from the screenshots presented in Fig. 4, the malicious applications tend to hide their malicious actions from the end-users by providing some entertainment content (e.g., games) which is one of the commonly used tactics ("Infected Fake Versions of Arcade Games on Google Play Threatened Players with Nasty Trojans," 2015; Villas-Boas, 2018). For the Android versions prior to Android 6.0, since the permissions that these applications need for their malicious actions are already granted during the installation time, they do not need user interactions to complete their aims. Therefore, upgrading the Android operating system at least to Android 6.0 is highly recommended for comprehensive permission control.

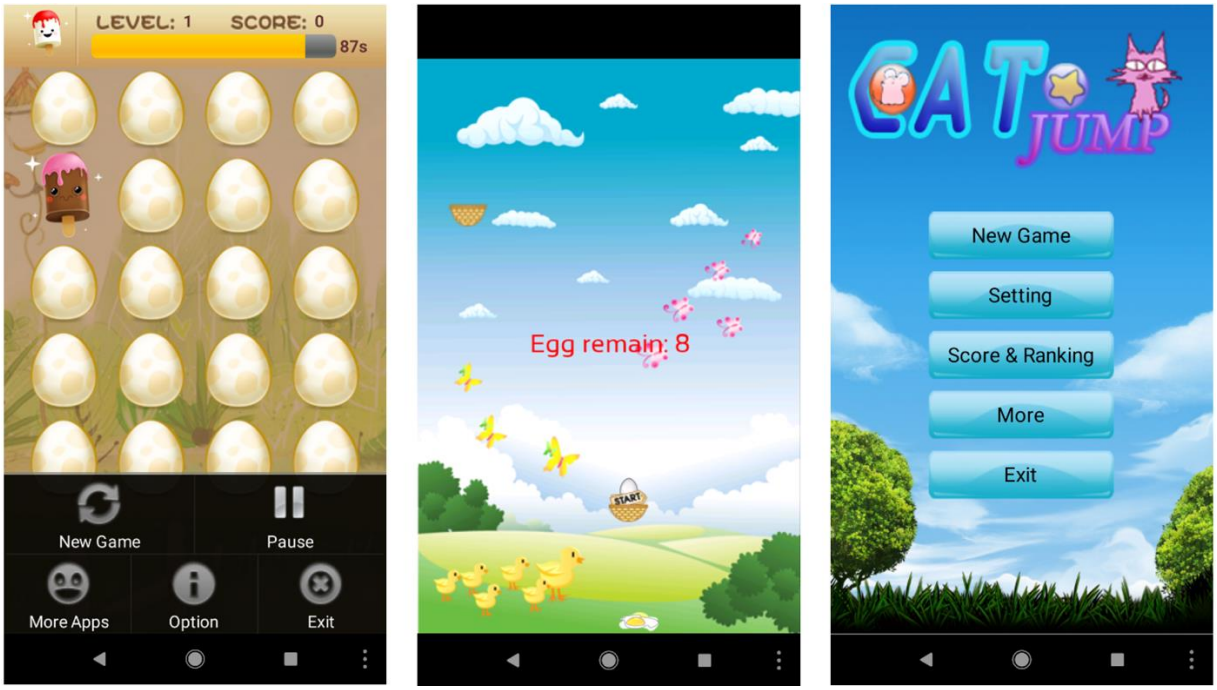

Figure 4. Sample screenshots that were captured during the executions of malicious applications.

An overview of the architecture of *Hybroid* is presented in Fig. 5. The whole operations for the analysis were carried out through the scripts, which were implemented using the Python programming language. The data of both the applications in the dataset and their analysis result was stored in the database, namely, *MongoDB*[8], which is the most popular NoSQL database ("DB-Engines Ranking - Popularity Ranking of Database Management Systems," 2019; Kaur & Rani, 2013; Violino, 2018) that provides quite better performance in terms of reading and writing data compared to the traditional SQL databases (Boicea, Radulescu, & Agapin, 2012; Ming Wu, 2015; Nyati, Pawar, & Ingle, 2013; Parker, Poe, & Vrbsky, 2013). For the dynamic analysis, the built-in emulator of Android SDK (*a.k.a.* Android Virtual Device) was preferred since it is

[8] https://www.mongodb.com

bundled with the official IDE for Android development namely *Android Studio*[9].

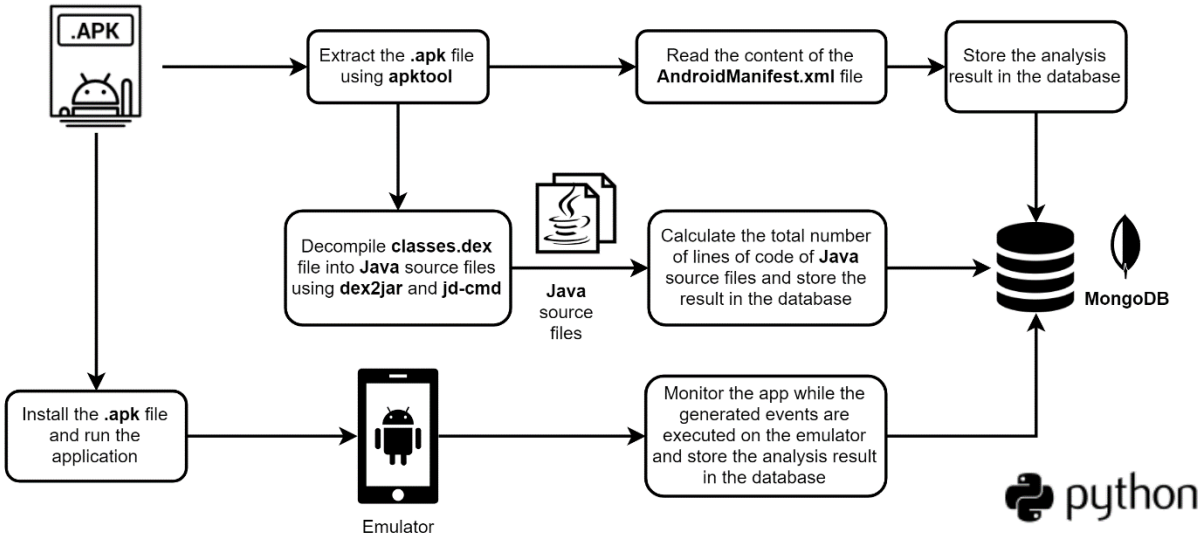

Figure 5. An overview of the architecture of Hybroid.

4. Experimental Results and Discussion

A dataset, that consists of 10,658 applications, was constructed from the widely-used, gold standard datasets in the literature. For the benign applications, the dataset that was constructed by (A.T. Kabakus & Dogru, 2018) was used which contains the applications from the various top charts (i.e. "Top Grossing Games", "Top Selling Games", "Music and Audio", etc.) in the *Play Store*. For the malicious applications, *Drebin* (Arp, Spreitzenbarth, Malte, Gascon, & Rieck, 2014), *F-droid*, and *Android Genome Project* (Zhou & Jiang, 2012) were intentionally preferred as they are widely used in the literature. The overview of the constructed dataset within the scope of *Hybroid* is listed in Table 8.

Table 8. The overview of the constructed dataset.

Dataset	Type	Number of Applications
***Drebin* (Arp et al., 2014)**	Malicious	5,373
F-droid	Malicious	1,123
***Android Genome Project* (Zhou & Jiang, 2012)**	Malicious	1,260
(A.T. Kabakus & Dogru, 2018)	Benign	2,902

The proposed framework, *Hybroid*, was utilized with various machine learning algorithms in order to reveal its effectiveness on Android malware detection. The performances of the classification systems based on supervised learning are generally evaluated by the confusion matrix, which greatly reflects the relationship between the classification results and actual values (Afonso et al., 2015; Kang et al., 2015; Narudin, Feizollah, Anuar, & Gani, 2016; Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014; X. Wang, Zhang, Su, & Li, 2017; Wu, Mao, Wei, Lee, & Wu, 2012). Hence, the *Hybroid*'s effectiveness was also evaluated by using the confusion matrix. The definitions of the metrics that the confusion matrix model defines are listed as follows:

[9] https://developer.android.com/studio/

(1) TP ($true\ positive$) means the number of malicious applications that are correctly classified, (2) TN ($true\ negative$) means the number of benign applications that are correctly classified, (3) FP ($false\ positive$) means the number of malicious applications that are classified as benign, and (4) FN ($false\ negative$) means the number of benign applications that are classified as malicious. The $accuracy$ is calculated as $Accuracy = (TP + TN) \div (TP + TN + FP + FN)$, the $recall$ ($aka\ true\ positive\ rate$ or $sensitivity$) is calculated as $Recall = TP \div (TP + FN)$, the $FP\ rate$ is calculated as $FP\ Rate = FP \div (TN + FP)$, the $precision$ is calculated as $Precision = TP \div (TP + FP)$, the $F - measure$ is calculated as $F - measure = (2 \times Recall \times Precision) \div (Recall + Precision)$, and the MCC ($Matthews\ Correlation\ Coefficient$) is calculated as $MCC = (TP \times TN - FP \times FN) \div \sqrt{[(TP + FP) \times (FN + TN) \times (FP + TN) \times (TP + FN)]}$. *Hybroid* was evaluated with thirteen widely-used machine learning algorithms, and the best $accuracy$ was calculated as high as 99.5% and the $FP\ rate$ was calculated as low as 0.8% when the system was trained with the *J48* algorithm as the confusion matrix of the experimental results is listed in Table 9. For all evaluations, a widely-used, open-source machine learning tool, namely, *WEKA* (Waikato Environment for Knowledge Analysis) ("Weka 3 - Data Mining with Open Source Machine Learning Software in Java," 2021), was utilized with the default configurations of the machine learning algorithms. The *arff* file format is one of the file formats that *WEKA* supports to load data from the filesystem. Hence, the data, which was stored in the database, was exported to an *arff* file through the implemented Python script.

Table 9. The confusion matrix of the experimental results of the evaluations of the machine learning algorithms.

Machine Learning Algorithm	Accuracy	FP Rate	Precision	Recall	F-measure	MCC
BayesNet	0.963	0.048	0.963	0.963	0.963	0.919
NaïveBayes	0.974	0.024	0.974	0.974	0.974	0.944
LogisticRegression	0.995	0.007	0.995	0.995	0.995	0.989
MultilayerPerceptron	0.993	0.011	0.993	0.993	0.993	0.985
SVM	0.985	0.027	0.985	0.985	0.985	0.967
kNN ($k = 1$)	0.979	0.031	0.979	0.979	0.979	0.954
AdaBoost	0.994	0.009	0.994	0.994	0.994	0.986
Bagging	0.994	0.009	0.994	0.994	0.994	0.987
LogitBoost	0.993	0.010	0.993	0.993	0.993	0.986
DecisionTable	0.995	0.009	0.995	0.995	0.995	0.988
J48	**0.995**	**0.008**	**0.995**	**0.995**	**0.995**	**0.988**
RandomForest	0.995	0.009	0.995	0.995	0.995	0.988
RandomTree	0.984	0.009	0.984	0.984	0.984	0.964

The attributes were evaluated with the *CfsSubsetEval* algorithm conjunction with the *GreedyStepwise* search algorithm, the selected attributes were found as follows, respectively: (1) The number of dangerous permissions *($numOfDangerousPermissions$),* (2) the total number of lines of code in the application's source code files *(loc),* (3) the niced processes executing in user mode *($niceCpu$),* (4) the number of received packets *($rxPackets$),* (5) the user-mode jiffies with child's *($cutime$),* and (6) the virtual memory size *(vss).* When the features were investigated through the experimental result as the analysis of each feature for both the benign and malicious applications is listed in Table 10, the following insights were deduced:

- Dangerous permissions were demanded by malicious applications more than benign applications which are reasonable since the malicious applications target hazardous actions that are mapped to dangerous permissions by the Android operating system (see Fig. 6).
- Activities and services were more widely-used in benign applications since their sole aim is to provide better service to the end-users while the malicious applications seek after to achieve their malicious actions. For the same reason, the average number of lines of code (loc) of benign applications is lots more than malicious applications (see Fig. 6).
- An unexpected insight was that despite that malicious applications do

not provide as many activities and services as benign applications (see Fig. 7), they consumed a similar size of memory. This can be explained as one of the aims of the malware is the abuse of system resources (Elish, Shu, Yao, Ryder, & Jiang, 2015).

Table 10. The statistics of the features for both benign and malicious applications.

Feature	Average for Benign Applications	Average for Malicious Applications
Number of dangerous permissions (*numOfDangerousPe*)	2.55	4.30
Number of activities (*numOfActivities*)	20.89	5.63
Number of services (*numOfServices*)	4.52	1.24
Number of lines of code (*loc*)	1,329,14	16,896
Niced processes executing in user mode (*niceCpu*)	3,782	2,486
Virtual memory size (*vss*) in MB	1,625	1,429

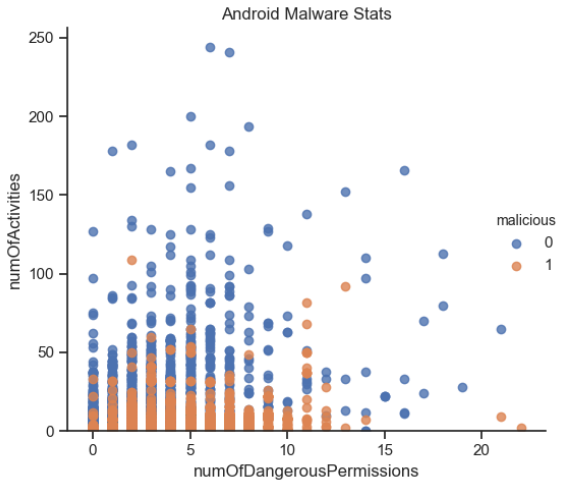

Figure 6. The plot of the usages of the number of activities (*numOfActivities*) and the number of dangerous permissions (*numOfDangerousPermissions*) by malicious and benign applications.

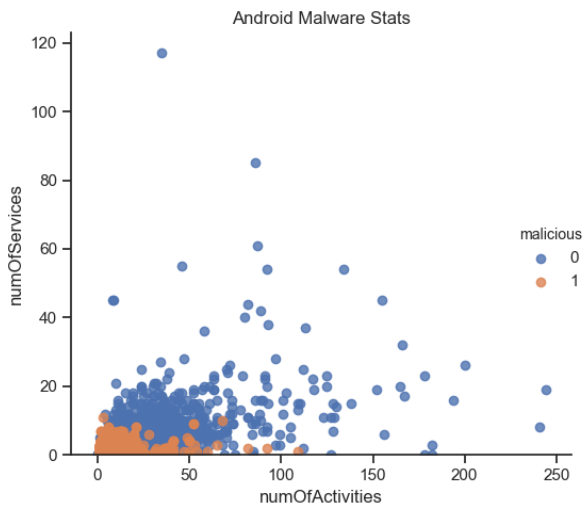

Figure 7. The plot of the usages of the number of activities ($numOfActivities$) and the number of services ($numOfServices$) by malicious and benign applications.

As the related work is described in Section 2, there are lots of studies regarding Android malware detection. Due to each proposed approach in the literature utilizes its own dataset because (1) fetching new apps from the *Play Store* makes the dataset unique and this is necessary as there does not exist both gold standard and up-to-date benign datasets, (2) some apps are being eliminated when their packaging architectures (e.g., *x86*, *ARM*) are not compatible with the employed one as this situation is also discussed in "*Known Limitations*". Therefore, we do not aim to directly compare the efficiency of the proposed study with the related work. Aside from providing an accuracy as high as 99.5%, the proposed framework utilizes an up-to-date Android version, namely, *Android Oreo*. As a natural result of using an up-to-date Android version, the new security mechanisms such as runtime permissions, and always denying mechanism are discussed with this study for the first time in the literature (to the best of our knowledge). The comparison of *Hybroid* with the related work in terms of the utilized classification algorithm and accuracy is listed in Table 11.

Table 11. The comparison of the related work.

Related Work	Classification Algorithm	Accuracy (%)
Drebin **(Arp et al., 2014)**	*SVM*	93.9
Zhang et al. **(Zhang, Duan, Yin, & Zhao, 2014)**	*Naïve Bayes*	93
Yerima et al. **(Yerima et al., 2013)**	*Bayesian Based Classifier*	92.1
McLaughlin et al. **(McLaughlin et al., 2017)**	*Convolutional Neural Network*	87
DroidDetector **(Yuan, Lu, & Xue, 2016)**	*Deep Belief Network*	96.8
SigPID **(J. Li et al., 2018)**	*Functional Tree*	93.62
FAMOUS **(A. Kumar et al., 2018)**	*Random Forest*	99
DroidDet **(Zhu et al., 2017)**	*Rotation Forest*	88.26
Sayfullina et al. **(Sayfullina et al., 2015)**	*Normalized Bernoulli Naïve Bayes*	91
DroidAPIMiner **(Aafer, Du, & Yin, 2013)**	*kNN*	99
SAMADroid **(Arshad, Shah, Wahid, Mehmood, & Song, 2018)**	*Random Forest*	99.07
DroidScribe **(Dash et al., 2016)**	*SVM* with *Conformal Prediction*	94
APK Auditor **(Abdullah Talha Kabakus et al., 2015)**	Own classifier based on *Logistic Regression*	88.28
DroidMat **(Wu et al., 2012)**	A combination of *K-means* and *kNN*	97.87
Hybroid	***J48***	**99.5**

Known Limitations

During the dynamic analysis, it was experienced that some applications were compiled for the *ARM* architecture. The emulator used within this study utilized the *Intel* architecture since (1) the *Android Oreo* images were not available for the *ARM* architecture at the time of writing, and (2) the images based on *x86* architecture are recommended by *Android Studio*. Hence, the applications, which are compiled for the *ARM* architecture, could not be analyzed. Also, some applications were not compatible with the API level of the emulator which is specifically selected to be a recent version of Android to handle the problem from an up-to-date perspective. Similarly, the applications in the utilized datasets which were packaged for the *ARM* architecture could not be dynamically analyzed due to the aforementioned reason. The number of applications in the constructed dataset was not increased further as it takes a relatively long time to complete the dynamic analysis. The proposed framework can be easily applied in an environment, which is powered by super-computers, to evaluate its effectiveness on huge datasets.

5. Conclusion

Android malware is still widespread despite the serious actions taken by Android. In this paper, a novel Android malware detection framework, namely, *Hybroid*, was proposed which combines both the static and dynamic analysis techniques in order to benefit from the advantages of both of them. Then *Hybroid* was trained with various machine learning algorithms and evaluated on a constructed dataset that consists of 10,658 applications in order to reveal its efficiency. According to the experimental result, the best accuracy was calculated as high as 99.5%

when *Hybroid* was utilized with the *J48* classification algorithm which outperforms the related state-of-the-art studies. The key contributions of this study are listed as follows:

- *Hybrid analysis. Hybroid* utilizes both static and dynamic analysis techniques in order to detect Android malware.
- *High accuracy and low FP rate. Hybroid*'s *accuracy* was calculated as high as 99.5% when it was evaluated on a large dataset that consists of 10,658 applications as it outperformed the related state-of-the-art studies. Similarly, *Hybroid*'s *FP rate* was calculated as low as 0.8%.
- *Key findings through experiments.* The proposed framework was evaluated on a large dataset that consists of a total of 10,658 applications, and the key findings through the experimental result were discussed in order to shed light on Android malware detection and inform the digital investigators.
- *An up-to-date perspective*. As a natural result of using an up-to-date Android version, namely, *Android Oreo*, the new security mechanisms, such as runtime permissions, and always denying mechanism, are discussed with this study for the first time in the literature (for the best of our knowledge).
- *Fully automated approach.* The proposed framework was fully automated which means no need for the human effort is required to carry out the analysis for all applications in the dataset. The input of *Hybroid* is the *apk* file of the application that is going to be analyzed. Then both the static and dynamic analysis is being carried out through the implemented Python software, and the analysis result is being stored in the database.

As future work, the construction of a dataset that contains more samples than the one used in this study is considered. Also, deep learning techniques can be integrated into the proposed malware detection framework.

Acknowledgments

The author would like to thank Computer Security Group – the University of Göttingen for sharing the *Drebin* dataset, and Zhou Y., Jiang X. for sharing the dataset of the *Android Malware Genome Project*.

References

Aafer, Y., Du, W., & Yin, H. (2013). DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. *9th International Conference on Security and Privacy in Communication Networks (SecureComm 2013)*, 86–103. Sydney, Australia. https://doi.org/10.1007/978-3-319-04283-1_6

Afonso, V. M., de Amorim, M. F., Grégio, A. R. A., Junquera, G. B., & de Geus, P. L. (2015). Identifying Android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, *11*(1), 9–17. https://doi.org/10.1007/s11416-014-0226-7

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2017). Improving Dynamic Analysis of Android Apps Using Hybrid Test Input Generation. *IEEE International Conference On Cyber Security And Protection Of Digital Services (Cyber Security 2017)*, 1–8. London, UK.

Android – Google Play Protect. (2021). Retrieved January 29, 2021, from Google website: https://www.android.com/play-protect/

Aresu, M., Ariu, D., Ahmadi, M., Maiorca,

D., & Giacinto, G. (2015). Clustering Android Malware Families by Http Traffic. *2015 10th International Conference on Malicious and Unwanted Software, MALWARE 2015*, 128–135. Fajardo, Puerto Rico. https://doi.org/10.1109/MALWARE.2015.7413693

Arp, D., Spreitzenbarth, M., Malte, H., Gascon, H., & Rieck, K. (2014). Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. *Symposium on Network and Distributed System Security (NDSS)*, 23–26. San Diego, California, USA.

Arshad, S., Ahmed, M., Shah, M. A., & Khan, A. (2016). Android Malware Detection & Protection: A Survey. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *7*(2), 463–475. https://doi.org/10.14569/IJACSA.2016.070262

Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., & Song, H. (2018). SAMADroid: A Novel 3-Level Hybrid Malware Detection Model for Android Operating System. *IEEE Access*, *6*, 4321–4339. https://doi.org/10.1109/ACCESS.2018.2792941

Backes, M., Gerling, S., Hammer, C., Maffei, M., Backes, M., Gerling, S., & Hammer, C. (2012). AppGuard - Real-time policy enforcement for third-party applications. Retrieved January 29, 2021, from Universitäts und Landesbibliothek Bonn website: http://sps.cs.uni-saarland.de/publications/monitor.pdf

Bae, C., & Shin, S. (2017). A collaborative approach on host and network level android malware detection. *Security and Communication Networks*, *9*(18), 5639–5650. https://doi.org/10.1002/sec.1723

Bao, L., Lo, D., Xia, X., & Li, S. (2017). Automated Android application permission recommendation. *Science China Information Sciences*, *60*(9), 1–17. https://doi.org/10.1007/s11432-016-9072-3

Bläsing, T., Batyuk, L., Schmidt, A. D., Camtepe, S. A., & Albayrak, S. (2010). An android application sandbox system for suspicious software detection. *5th IEEE International Conference on Malicious and Unwanted Software (Malware 2010)*, 55–62. Nancy, France: IEEE. https://doi.org/10.1109/MALWARE.2010.5665792

Boicea, A., Radulescu, F., & Agapin, L. I. (2012). MongoDB vs Oracle - Database comparison. *Proceedings of 3rd International Conference on Emerging Intelligent Data and Web Technologies, EIDWT 2012*, 330–335. Bucharest, Romania. https://doi.org/10.1109/EIDWT.2012.32

Bowden, T., Bauer, B., Nerin, J., Feng, S., & Seibold, S. (2018). The /proc Filesystem. Retrieved January 29, 2021, from https://github.com/torvalds/linux/blob/master/Documentation/filesystems/proc.txt

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: Behavior-Based Malware Detection System for Android. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices - SPSM '11*, 1–11. Chicago, IL, USA. https://doi.org/10.1145/2046614.2046619

Burke, D. (2017). Android: celebrating a big milestone together with you. Retrieved January 29, 2021, from Google website: https://www.blog.google/products/android/2bn-milestone/

Cacek, J. (2018). kwart/jd-cmd: Command line Java Decompiler. Retrieved January 29, 2021, from https://github.com/kwart/jd-cmd

Canfora, G., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2015). Detecting Android malware using sequences of system calls. *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile - DeMobile 2015*, 13–20. Bergamo, Italy.

https://doi.org/10.1145/2804345.2804349

Chandramohan, M., & Tan, H. B. K. (2012). Detection of Mobile Malware in the Wild. *Computer*, *45*(9), 65–71. https://doi.org/10.1109/MC.2012.36

Cunningham, E. (2017). Keeping you safe with Google Play Protect. Retrieved January 29, 2021, from Google website: https://blog.google/products/android/google-play-protect/

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., & Cavallaro, L. (2016). DroidScribe: Classifying Android Malware Based on Runtime Behavior. *Proceedings - 2016 IEEE Symposium on Security and Privacy Workshops, SPW 2016*, 252–261. San Jose, CA, USA. https://doi.org/10.1109/SPW.2016.25

DB-Engines Ranking - popularity ranking of database management systems. (2019). Retrieved January 29, 2021, from DB-Engines website: https://db-engines.com/en/ranking

Di Cerbo, F., Girardello, A., Michahelles, F., & Voronkova, S. (2011). Detection of malicious applications on android OS. *4th International Workshop on Computational Forensics, IWCF 2010, November 11, 2010 - November 12, 2010, 6540 LNCS*, 138–149. https://doi.org/10.1007/978-3-642-19376-7_12

Dini, G., Martinelli, F., Saracino, A., & Sgandurra, D. (2012). MADAM: A Multi-level Anomaly Detector for Android Malware. In I. Kotenko & V. Skormin (Eds.), *Computer Network Security* (pp. 240–253). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33704-8

Elish, K. O., Shu, X., Yao, D., Ryder, B. G., & Jiang, X. (2015). Profiling user-trigger dependence for Android malware detection. *Computers and Security*, *49*, 255–273. https://doi.org/10.1016/j.cose.2014.11.001

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., Mcdaniel, P., & Sheth, A. N. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, 393–407. Vancouver, BC, Canada.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On Lightweight Mobile Phone Application Certification. *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, 235–245. Chicago, Illinois, USA. https://doi.org/10.1145/1653662.1653691

Fan, W., Sang, Y., Zhang, D., Sun, R., & Liu, Y. (2017). DroidInjector: A process injection-based dynamic tracking system for runtime behaviors of Android applications. *Computers and Security*, *70*, 224–237. https://doi.org/10.1016/j.cose.2017.06.001

Faruki, P., Ganmoor, V., Laxmi, V., Gaur, M. S., & Bharmal, A. (2013). AndroSimilar: Robust Statistical Feature Signature For Android Malware Detection. *Proceedings of the 6th International Conference on Security of Information and Networks - SIN '13*, 1–8. https://doi.org/10.1145/2523514.2523539

Faruki, P., Zemmari, A., Gaur, M. S., Laxmi, V., & Conti, M. (2016). MimeoDroid: Large Scale Dynamic App Analysis on Cloned Devices via Machine Learning Classifiers. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016*, 60–65. https://doi.org/10.1109/DSN-W.2016.33

Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*, 627–638. New York, New York, USA: ACM Press. https://doi.org/10.1145/2046707.2046779

Fuchs, A. P., Chaudhuri, A., & Foster, J. S. (2009). *SCanDroid: Automated Security Certification of Android Applications*.

https://doi.org/10.1.1.164.6899

Gadhiya, S., Bhavsar, K., & Student, P. D. (2013). Techniques for Malware Analysis. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3*(4), 972–975.

Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012). AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. *TRUST'12 Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, *7344 LNCS*, 291–307. Vienna, Austria. https://doi.org/10.1007/978-3-642-30921-2_17

Grace, M., Zhou, Y., Wang, Z., & Jiang, X. (2012). Systematic Detection of Capability Leaks in Stock Android Smartphones. *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, 1–15. San Diego, California, USA.

Grace, M., Zhou, Y., Zhang, Q., Zou, S., & Jiang, X. (2012). RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services - MobiSys '12*, 281–294. Low Wood Bay, Lake District, United Kingdom: ACM Press. https://doi.org/10.1145/2307636.2307663

IDC - Smartphone Market Share. (2019). Retrieved January 29, 2021, from IDC website: https://www.idc.com/promo/smartphone-market-share/os

Infected Fake Versions of Arcade Games on Google Play Threatened Players with Nasty Trojans. (2015). Retrieved January 29, 2021, from ESET website: https://www.eset.com/int/about/newsroom/press-releases/announcements/infected-arcade-games-trojan-dropper/

Kabakus, A.T., & Dogru, I. A. (2018). An in-depth analysis of Android malware using hybrid techniques. *Digital Investigation*, *24*, 25–33. https://doi.org/10.1016/j.diin.2018.01.001

Kabakus, Abdullah Talha, Dogru, I. A., & Cetin, A. (2015). APK Auditor: Permission-based Android malware detection system. *Digital Investigation*, *13*, 1–14. https://doi.org/10.1016/j.diin.2015.01.001

Kang, H., Jang, J. W., Mohaisen, A., & Kim, H. K. (2015). Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, *2015*, 1–9. https://doi.org/10.1155/2015/479174

Karbab, E. M. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, *24*, S48–S59. https://doi.org/10.1016/j.diin.2018.01.007

Kaur, K., & Rani, R. (2013). Modeling and querying data in NoSQL databases. *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, 1–7. Santa Clara, CA, USA: IEEE. https://doi.org/10.1109/BigData.2013.6691765

Kayikci, S. (2018). A Deep Learning Method for Passing Completely Automated Public Turing Test. *3rd International Conference on Computer Science and Engineering (UBMK 2018)*, 41–44. Sarajevo, Bosnia-Herzegovina: IEEE. https://doi.org/10.1109/UBMK.2018.8566318

Kelley, P. G., Consolvo, S., Cranor, L. F., Jung, J., Sadeh, N., & Wetherall, D. (2012). A Conundrum of Permissions: Installing Applications on an Android Smartphone. *Proceedings of the 16th International Conference on Financial Cryptography and Data Security (FC '12)*, 68–79. Kralendijk, Bonaire: Springer. https://doi.org/10.1007/978-3-642-34638-5

Kim, T., Kang, B., Rho, M., Sezer, S., & Im, E. G. (2019). A multimodal deep learning

method for android malware detection using various features. *IEEE Transactions on Information Forensics and Security*, *14*(3), 773–788. https://doi.org/10.1109/TIFS.2018.2866319

King, J., Lampinen, A., & Smolen, A. (2011). Privacy: is there an app for that? *Proceedings of the Seventh Symposium on Usable Privacy and Security (SOUPS '11)*, 1–20. New York, NY, USA: ACM Press. https://doi.org/10.1145/2078827.2078843

Kumar, A., Kuppusamy, K. S., & Aghila, G. (2018). FAMOUS: Forensic Analysis of MObile Using Scoring of application permission. *Future Generation Computer Systems*, *83*, 158–172. https://doi.org/10.1016/j.future.2018.02.001

Kumar, M. (2017). Beware! New Android Malware Infected 2 Million Google Play Store Users. Retrieved January 29, 2021, from The Hacker News website: http://thehackernews.com/2017/04/android-malware-playstore.html

Li, B., Zhang, Y., Li, J., Yang, W., & Gu, D. (2018). APPSPEAR: Automating the hidden-code extraction and reassembling of packed android malware. *Journal of Systems and Software*, *140*, 3–16. https://doi.org/10.1016/j.jss.2018.02.040

Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., & Ye, H. (2018). Significant Permission Identification for Machine-Learning-Based Android Malware Detection. *IEEE Transactions on Industrial Informatics*, *14*(7), 3216–3225. https://doi.org/10.1109/TII.2017.2789219

Liang, S., & Du, X. (2014). Permission-combination-based scheme for Android mobile malware detection. *2014 IEEE International Conference on Communications (ICC)*, 2301–2306. Sydney, Australia: IEEE. https://doi.org/10.1109/ICC.2014.6883666

Liu, L., Yan, G., Zhang, X., & Chen, S. (2009). Virusmeter: Preventing your cellphone from spies. *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID '09)*, 244–264. https://doi.org/10.1007/978-3-642-04342-0_13

Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., & Stavrou, A. (2012). A whitebox approach for automated security testing of Android applications on the cloud. *7th International Workshop on Automation of Software Test (AST 2012)*, 22–28. Zurich, Switzerland: IEEE Press. https://doi.org/10.1109/IWAST.2012.6228986

Massarelli, L., Aniello, L., Ciccotelli, C., Querzoni, L., Ucci, D., & Baldoni, R. (2017). Android Malware Family Classification Based on Resource Consumption over Time. *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, 31–38. Fajardo, PR, USA. Retrieved from http://arxiv.org/abs/1709.00875

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., … Joon Ahn, G. (2017). Deep Android Malware Detection. *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY '17*, 301–308. Scottsdale, Arizona, USA. https://doi.org/10.1145/3029806.3029823

Milano, D. T. (2018). AndroidViewClient. Retrieved January 29, 2021, from https://github.com/dtmilano/AndroidViewClient

Ming Wu, C. (2015). Comparisons Between MongoDB and MS-SQL Databases on the TWC Website. *American Journal of Software Engineering and Applications*, *4*(2), 35–41. https://doi.org/10.11648/j.ajsea.20150402.12

Morris, D. Z. (2017). Android Malware Judy' Hits As Many As 36.5 Million Phones. Retrieved January 29, 2021, from Fortune website: http://fortune.com/2017/05/28/android-malware-judy/

Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of static analysis for malware detection. *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, 421–430. Miami Beach, FL, USA. https://doi.org/10.1109/ACSAC.2007.21

Mylonas, A., Kastania, A., & Gritzalis, D. (2013). Delegate the smartphone user? Security awareness in smartphone platforms. *Computers and Security*, *34*, 47–66. https://doi.org/10.1016/j.cose.2012.11.004

Narudin, F. A., Feizollah, A., Anuar, N. B., & Gani, A. (2016). Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, *20*(1), 343–357. https://doi.org/10.1007/s00500-014-1511-6

Nyati, S. S., Pawar, S., & Ingle, R. (2013). Performance evaluation of unstructured NoSQL data over distributed framework. *Proceedings of the 2013 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2013*, 1623–1627. https://doi.org/10.1109/ICACCI.2013.6637424

Parker, Z., Poe, S., & Vrbsky, S. V. (2013). Comparing NoSQL MongoDB to an SQL DB. *Proceedings of the 51st ACM Southeast Conference on - ACMSE '13*, 1–6. https://doi.org/10.1145/2498328.2500047

Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., … Molloy, I. (2012). Using Probabilistic Generative Models for Ranking Risks of Android Apps. *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, 241–252. Raleigh, North Carolina, USA. https://doi.org/10.1145/2382196.2382224

Permissions overview | Android Developers. (2019). Retrieved January 29, 2021, from Google website: https://developer.android.com/guide/topics/permissions/overview#normal_permissions

Popper, B. (2017). Google announces over 2 billion monthly active devices on Android. Retrieved January 29, 2021, from The Verge website: https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users

Portokalidis, G., Homburg, P., Anagnostakis, K., & Bos, H. (2010). Paranoid Android: Versatile Protection For Smartphones. *Annual Computer Security Applications Conference (ACSAC)*, 347–356. Austin, Texas, USA. https://doi.org/10.1145/1920261.1920313

pxb1988/dex2jar: Tools to work with android .dex and java .class files. (2018). Retrieved January 29, 2021, from https://github.com/pxb1988/dex2jar

Rastogi, V., Chen, Y., & Enck, W. (2013). AppsPlayground : Automatic Security Analysis of Smartphone Applications. *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, 209–220. San Antonio, Texas, USA. https://doi.org/10.1145/2435349.2435379

Rhode, M., Burnap, P., & Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. *Computers and Security*, *77*, 578–594. https://doi.org/10.1016/j.cose.2018.05.010

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P. G., & Álvarez, G. (2012). PUMA: Permission usage to detect malware in android. *International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*, 289–298. Ostrava, Czech Republic: Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33018-6_30

Sato, R., Chiba, D., & Goto, S. (2013). Detecting Android Malware by Analyzing Manifest Files. *Proceedings of the Asia-Pacific Advanced Network 2013 (APAN '13)*, 23–31. Kaist, Daejeon, Korea. https://doi.org/10.7125/APAN.36.4

Sayfullina, L., Eirola, E., Komashinsky, D., Palumbo, P., Miche, Y., Lendasse, A., & Karhunen, J. (2015). Efficient detection of

zero-day android malware using normalized bernoulli naive bayes. *Proceedings of The14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2015)*, 198–205. Helsinki, Finland. https://doi.org/10.1109/Trustcom.2015.375

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). "Andromaly": A behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, *38*, 161–190. https://doi.org/10.1007/s10844-010-0148-x

Shabtai, A., Tenenboim-Chekina, L., Mimran, D., Rokach, L., Shapira, B., & Elovici, Y. (2014). Mobile malware detection through analysis of deviations in application network behavior. *Computers & Security*, *43*, 1–18. https://doi.org/10.1016/j.cose.2014.02.009

Singh, P., Tiwari, P., & Singh, S. (2016). Analysis of Malicious Behavior of Android Apps. *Procedia Computer Science*, *79*, 215–220. https://doi.org/10.1016/j.procs.2016.03.028

SophosLabs 2018 Malware Forecast. (2018). Retrieved January 29, 2021, from Sophos website: https://www.sophos.com/en-us/en-us/medialibrary/PDFs/technical-papers/malware-forecast-2018.pdf

Spreitzenbarth, M., Freiling, F. C., Echtler, F., Schreck, T., & Hoffmann, J. (2013). Mobile-sandbox: Having a Deeper Look into Android Applications. *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*, 1808–1815. Coimbra, Portugal: ACM. https://doi.org/10.1145/2480362.2480701

Stefanko, L. (2018). Android Trojan steals money from PayPal accounts even with 2FA on. Retrieved January 29, 2021, from ESET website: https://www.welivesecurity.com/2018/12/11/android-trojan-steals-money-paypal-accounts-2fa/

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, *41*(4 PART 1), 1104–1117. https://doi.org/10.1016/j.eswa.2013.07.106

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Ribagorda, A. (2014). Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys and Tutorials*, *16*(2), 961–987. https://doi.org/10.1109/SURV.2013.101613.00077

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., & Cavallaro, L. (2017). The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys*, *49*(4), 1–41. https://doi.org/10.1145/3017427

Tang, W., Jin, G., He, J., & Jiang, X. (2011). Extending android security enforcement with a security distance model. *Proceedings of the 2011 International Conference on Internet Technology and Applications (ITAP 2011)*, 1–4. https://doi.org/10.1109/ITAP.2011.6006288

The Mobile Economy 2018. (2018). Retrieved January 29, 2021, from https://www.gsma.com/mobileeconomy/wp-content/uploads/2018/02/The-Mobile-Economy-Global-2018.pdf

Tong, F., & Yan, Z. (2017). A hybrid approach of mobile malware detection in Android. *Journal of Parallel and Distributed Computing*, *103*, 22–31. https://doi.org/10.1016/j.jpdc.2016.10.012

Villas-Boas, A. (2018). Google removed 13 games from the Play Store for containing malware. Retrieved January 29, 2021, from Business Insider website: https://www.businessinsider.com/google-play-store-game-apps-removed-malware-2018-11

Violino, B. (2018). How to choose the right NoSQL database. Retrieved January 29, 2021, from InfoWorld website:

https://www.infoworld.com/article/3260184/nosql/how-to-choose-the-right-nosql-database.html

Wang, C., Li, Z., Mo, X., Yang, H., & Zhao, Y. (2017). An android malware dynamic detection method based on service call co-occurrence matrices. *Annals of Telecommunications*, *72*(9–10), 1–9. https://doi.org/10.1007/s12243-017-0580-9

Wang, X., Zhang, D., Su, X., & Li, W. (2017). Mlifdect: Android malware detection based on parallel machine learning and information fusion. *Security and Communication Networks*, *2017*, 1–15. https://doi.org/10.1155/2017/6451260

Wang, Y., Zheng, J., Sun, C., & Mukkamala, S. (2013). Quantitative security risk assessment of Android permissions and applications. In L. Wang & B. Shafiq (Eds.), *27th Data and Applications Security and Privacy (DBSec)* (pp. 226–241). Newark, NJ, USA: Springer. https://doi.org/10.1007/978-3-642-39256-6_15

Wei, L., Luo, W., Weng, J., Zhong, Y., Zhang, X., & Yan, Z. (2017). Machine Learning-Based Malicious Application Detection of Android. *IEEE Access*, *5*, 25591–25601. https://doi.org/10.1109/ACCESS.2017.2771470

Wei, X., Gomez, L., Neamtiu, I., & Faloutsos, M. (2012). Malicious Android Applications in the Enterprise: What Do They Do and How Do We Fix It? *ICDEW '12 Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, 251–254. Arlington, Virginia, USA: IEEE.

Weka 3 - Data Mining with Open Source Machine Learning Software in Java. (2021). Retrieved January 29, 2021, from https://www.cs.waikato.ac.nz/ml/weka/

Who writes Linux? Almost 10,000 developers. (2013). Retrieved January 29, 2021, from ZDNet website: https://www.zdnet.com/article/who-writes-linux-almost-10000-developers/

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., & Wu, K.-P. (2012). DroidMat: Android Malware Detection through Manifest and API Calls Tracing. *2012 Seventh Asia Joint Conference on Information Security*, 62–69. Minato, Tokyo, Japan. https://doi.org/10.1109/AsiaJCIS.2012.18

Xue, Y., Meng, G., Liu, Y., Tan, T. H., Chen, H., Sun, J., & Zhang, J. (2017). Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. *IEEE Transactions on Information Forensics and Security*, *12*(7), 1529–1544. https://doi.org/10.1109/TIFS.2017.2661723

Yang, M., Wang, S., Ling, Z., Liu, Y., & Ni, Z. (2017). Detection of malicious behavior in android apps through API calls and permission uses analysis. *Concurrency and Computation: Practice and Experience*, *29*(19), 1–13. https://doi.org/10.1002/cpe.4172

Yerima, S. Y., Sezer, S., McWilliams, G., & Muttik, I. (2013). A New Android Malware Detection Approach Using Bayesian Classification. *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, 121–128. Barcelona, Spain: IEEE. https://doi.org/10.1109/AINA.2013.88

Yu, J., Huang, Q., & Yian, C. H. (2016). DroidScreening: a practical framework for real-world Android malware analysis. *Security and Communication Networks*, *9*(11), 1435–1449. https://doi.org/10.1002/sec.1430

Yuan, Z., Lu, Y., & Xue, Y. (2016). DroidDetector: Android Malware Characterization and Detection Using Deep Learning. *Tsinghua Science and Technology*, *21*(1), 114–123. https://doi.org/10.1109/TST.2016.7399288

Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. *Proceedings of the*

2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14), 1105–1116. https://doi.org/10.1145/2660267.2660359

Zhao, M., Ge, F., Zhang, T., & Yuan, Z. (2011). AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android. *Communications in Computer and Information Science*, *243 CCIS*, 158–166. https://doi.org/10.1007/978-3-642-27503-6_22

Zheng, M., Sun, M., & Lui, J. C. S. (2013). DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 163–171. Melbourne, Victoria, Australia. https://doi.org/10.1109/TrustCom.2013.25

Zheng, M., Sun, M., & Lui, J. C. S. (2014). DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. *IWCMC 2014 - 10th International Wireless Communications and Mobile Computing Conference*, 128–133. Nicosia, Cyprus. https://doi.org/10.1109/IWCMC.2014.6906344

Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, 95–109. San Francisco, CA, USA: IEEE. https://doi.org/10.1109/SP.2012.16

Zhu, H. J., You, Z. H., Zhu, Z. X., Shi, W. L., Chen, X., & Cheng, L. (2017). DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, *272*, 638–646. https://doi.org/10.1016/j.neucom.2017.07.030

Appendix

Appendix 1. The content of a sample */proc/net/dev* file

```
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
radio0:    7489      56    0    0    0     0          0         0    30349     194    0    0    0     0       0          0
 wlan0: 85773067   58825    0    0    0     0          0         0  2567199   28267    0    0    0     0       0          0
  sit0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
    lo:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
hwsim0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
```

Appendix 2. The content of a sample */proc/[PID]/stat* file

8208 (.android.chrome) S 1495 1495 0 0 -1 1077961024 9029 21 346 0 49 52 0 0 16 -4 39 0 264240 1566953472 36326 4294967295 1 1 0 0 0 0 4612 0 1073775864 4294967295 0 0 17 3 0 0 23 0 0 0 0 0 0 0 0 0 0

Appendix 3. The content of a sample */proc/[PID]/statm* file

```
382557 36260 31201 4 0 38152 0
```