Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset

Article *in* Information Fusion · December 2018

DOI: 10.1016/j.inffus.2018.12.006

Android malware detection through hybrid features fusion and ensemble classifiers: the AndroPyTool framework and the OmniDroid dataset

Alejandro Martín[a], Raúl Lara-Cabrera[b], David Camacho[a,*]

[a] *Computer Science Department*
Universidad Autónoma de Madrid, 28049 Spain
[b] *Departamento de Sistemas Informáticos*
Universidad Politécnica de Madrid, 28031 Spain

Abstract

Cybersecurity has become a major concern for society, mainly motivated by the increasing number of cyber attacks and the wide range of targeted objectives. Due to the popularity of smartphones and tablets, Android devices are considered an entry point in many attack vectors. Malware applications are among the most used tactics and tools to perpetrate a cyber attack, so it is critical to study new ways of detecting them. In these detection mechanisms, machine learning has been used to build classifiers that are effective in discerning if an application is malware or benignware. However, training such classifiers require big amounts of labelled data which, in this context, consist of categorised malware and benignware Android applications represented by a set of features able to describe their behaviour. For that purpose, in this paper we present *OmniDroid*, a large and comprehensive dataset of features extracted from 22,000 real malware and goodware samples, aiming to help anti-malware tools creators and researchers when improving, or developing, new mechanisms and tools for Android malware detection. Furthermore, the characteristics of the dataset make it suitable to be used as a benchmark dataset to test classification and clustering algorithms or new representation techniques, among others. The dataset has been released under a *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License* and was built using AndroPyTool, our automated framework for dynamic and static analysis of Android applications. Finally, we test a set of ensemble classifiers over this dataset and propose a malware detection approach based on the fusion of static and dynamic features through the combination of ensemble classifiers. The experimental results show the feasibility and potential usability (for the machine learning, soft computing and cyber security communities) of our automated framework and the publicly available dataset.

Keywords: Malware analysis, Android, Hybrid features fusion, Malware dataset

1. Introduction

Due to the presence of technology in all areas of our daily lives, cyber security has become one of the main concerns to be addressed by the society as a whole. In recent years, there has been a large number of attacks and, what is even more remarkable, to a wide variety of objectives. Some recent well-known examples include denial of service attacks such as that performed by the Mirai botnet [1] and a massive data hijacking led by the ransom-ware Wannacry [2].

Furthermore, mobile devices are everywhere nowadays due to their popularity. Even in big companies this has been noticed, thus implementing new policies such as BYOD (Bring Your Own Device) and increasing the number of telecommuting employees. But, on the other hand, this blurs the perimeter security even more in these companies. Mobile devices can be considered as an entry point in any attack vector since the security measures are not so developed as in PCs. Hence, it is required to research new techniques for the automatic detection of malware for mobile devices, especially those that use Android operating system, since it represents over the 80% of the market share compared to iOS (around 15%), according to the Worldwide Quarterly Mobile Phone Tracker [3].

Cyber attacks manage to produce unprecedented levels of disruption, where attackers usually leverage diverse tools and tactics, such as zero-day vulnerabilities and malware [4]. This situation makes malware detection techniques worth studying and improving, in order to prevent and/or mitigate the effects of cyber attacks. Machine learning techniques can help to satisfy this demand, building classifiers that discern whether a precise Android application is malware or benignware. Algorithms such as Decision Trees [5], Support-Vector Machines [6] and Naive Bayes [7], to name a few, are able to build such classifiers. Going further, ensemble methods for machine learning [8]

[*] *Corresponding author*
Email addresses: `alejandro.martin@uam.es` (Alejandro Martín), `raul.lara@upm.es` (Raúl Lara-Cabrera), `david.camacho@uam.es` (David Camacho)

aim at effectively integrating many kinds of classification methods and learners to benefit from each ones advantages and overcome their individual drawbacks, hence improving the overall performance of the classification.

Nevertheless, machine learning techniques require large datasets of representative features extracted from real samples, which defines one of the goals of this paper: to present a comprehensive dataset of *dynamic* and *static* features from Android applications called *OmniDroid*. This dataset can help other researchers to improve and develop new automatic malware detection techniques for Android devices. At the same time, the characteristics of this dataset make it suitable to be employed as a benchmark dataset to apply and test different algorithms and techniques, such as classification, clustering, association rule learning, pattern recognition or even to test representation learning algorithms. In order to make the *OmniDroid* dataset easy to use, all the data are provided in JSON and CSV formats and are publicly available.[1]

The *OmniDroid* dataset has been built using *AndroPyTool* [9, 10], an automated open source tool for dynamic and static analysis of Android applications. Hence, another goal of this paper is to present *AndroPyTool* in depth and to describe its functionality.

The main contributions of this paper can be summarised as follows:

- *AndroPyTool*, a new tool for the automatic extraction of both static and dynamic features from sets of Android applications, is presented.

- The aforementioned tool has been used to generate a dataset (called *OmniDroid*) of static and dynamic features extracted from Android benignware and malware samples. The dataset is publicly available under a *Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License* [11].

- A thorough analysis showing statistics regarding the differences found between the two categories of samples and among the samples belonging to the same set.

- The performance of several ensemble classifiers from the state of the art have been studied to assess the feasibility of the features selected to build new detection or classification models based on ensemble techniques. It also illustrates the potential advantages of using the tool and the dataset when building ensemble methods to detect and classify Android malware.

- Finally, an Android malware detection approach, based on the fusion of static and dynamic features through the combination of an ensemble of classifiers following a voting scheme, is presented.

2. Basics on Android malware detection

Malware, in its different forms, represents a major issue affecting different platforms, from personal computers to smartphones or to the Internet of Things (IoT). From the appearance of the first virus designed for computers in the 70's, trojans, viruses or spyware, among others, have been developing different shapes to deal with the countermeasures imposed by operating systems and anti-virus engines. This has led to a race where security experts are always pursuing black hat hackers. To tackle this problem, malware detection tools try to extract and model the behaviour of a suspicious sample, which is compared against benign or malicious patterns in order to make a decision. This section describes the most used features capable of representing these behaviours and how they are used in the literature.

2.1. Basic features for Android malware analysis

In order to model the behaviour of a benign or malicious app, it is required to establish a representation able to describe in depth its actions and purposes. Two approaches to conduct this modelling process are possible: a static or a dynamic analysis, presenting different procedures to be performed, at the same time that entail a series of advantages and disadvantages. In order to characterise the behaviour of a given sample following static analysis techniques, it is possible to inspect the package, to decompile the code or to access the different files contained in the package (i.e. the Android Manifest). This allows to gather a set of relevant and useful features, such as a list of API calls invoked throughout the code or the set of Android permissions required in order to deploy the whole functionality of the sample.

In contrast, dynamic analysis opts for capturing the actions that are actually triggered by the suspicious sample, leveraging an emulator or even a physical device to run the app while a monitoring agent captures a series of indicators, such as hardware components accessed, network traffic or system calls invoked. One of the main benefits of following a dynamic approach lies in its ability to capture events invoked in obfuscated sections of code or included in code dynamically loaded. In these two examples, static analysis faces a major barrier.

Detection and classification tools leverage groups of specific features, whether static, dynamic or combinations of both, through different representation techniques, such as histograms, graphs or Markov models. This subsection describes the most employed and cited features in the literature and how they are handled when building detection and classification tools. A summary of the different state-of-the-art approaches is provided in Table 1.

API calls are within the most used features. Whether static or dynamically extracted, they allow to model the behaviour of a sample and to characterise the actions it can take. For instance, DroidMat [24] performs API Calls

[1] `https://aida.ii.uam.es/datasets/`

Detection/ classification method	API calls	Files access	Intents	Permissions	Network data	Hardware	Native Code	Hidden files/ dynamic loading	ICC	Metainf.	Opcodes	System commands	Strings	Taint analysis
ADROIT[12]				✓						✓				
Andro-Dumpsys[13]	✓		✓	✓								✓		
AndroDialysis[14]			✓	✓										
Andromaly[15]					✓	✓								
Apposcopy[16]	✓													✓
Dendroid[17]											✓			
DREBIN[18]			✓	✓		✓				✓				
DroidAnalytics[19]	✓							✓			✓			
DroidAPIMiner[20]	✓													
DroidDet[21]	✓		✓	✓										
DroidFusion[22]	✓		✓	✓				✓				✓		
DroidLegacy[23]	✓													
DroidMat[24]	✓		✓	✓		✓			✓					
DroidMiner[25]	✓		✓	✓										
DroidScribe[26]		✓			✓			✓	✓					
DroidSieve[27]	✓		✓	✓			✓	✓		✓	✓		✓	
Gascon et al.[28]	✓													
ICCDetector[29]			✓						✓	✓				
Madam[30]	✓	✓		✓						✓				
Manilyzer[31]			✓							✓				
Marvin[32]	✓	✓	✓	✓	✓			✓		✓				
MAST[33]			✓	✓			✓	✓						
Peiravian et al.[34]	✓			✓										
RevealDroid[35]	✓		✓											✓
Sheen et al.[36]	✓			✓										
Wang et al.[37]	✓		✓	✓		✓								
Yerima et al.[38]	✓			✓				✓				✓		
Yerima et al.[39]	✓			✓								✓		
Zhang et al.[40]	✓													✓
OmniDroid	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓

Table 1: Relation of features extracted and used by different Android malware detection and classification state-of-the-art approaches. The bottom lines indicates the set of features that the presented dataset OmniDroid contains. In the case of the detection of files accessed, the use of dynamic code loading techniques or information related to network connections, these are extracted during the dynamic analysis performed with DroidBox.

tracing from the different app components. DroidAPIMiner [20] also focuses on such API calls that are considered critical by the authors and include the arguments used when the call is invoked. API calls have also been used in the form of call graphs [28].

A relation of the Intents declared by an application can also be deemed as a key information source to categorise the behaviour of a sample. As in the case of API calls, a list of actions defined by Intents can be static or dynamically extracted. There are many examples in the literature studying this feature. In MAST [33], different indicators of the application functionality, including Intent actions, are analysed based on the assumption that these factors differ from benign samples. This feature has also been used to group malicious samples based on similarity [13]. For further information, the effectiveness of Intent actions for revealing malicious behavioural patterns has been evaluated by Feizollah et al. [14].

Android permissions are one of the most relevant characteristics in the Android malware detection scenario, even though they cannot offer a detailed description of the intentions that a suspicious application could take. Malware detection models based on machine learning algorithms have been trained with datasets of permissions combined with API calls [34]. In combination with many other features, such as filtered intents, network information or data regarding the use of hardware components, Support Vector Machines are trained in Drebin [18] for malware family classification. With a special focus on detecting obfuscated malware, DroidSieve [27] combines permissions with a list of invoked components or API calls, among others.

From a more general point of view, other methods base their malware detection mechanisms solely on meta-information provided by the developers. Such is the case of ADROIT [12], a system that trains machine learning algorithms with a set of features extracted from the Android Manifest and performs a text mining process on the description text of the application. Another example is Manilyzer [31], focused on using the information which can be gathered from the Android Manifest files to train machine learning algorithms.

Other possibilities to face this problem, involving statically features extracted as well, are based on the employment of opcodes or system commands. Droid Analytics [19] is a system aimed at assisting to retrieve opcode level information from Android malware. A study on the effectiveness of opcodes for malware family classification can also be found in the literature [41]. System commands have been used in combination with a parallel machine learning classifier [39] or Bayesian classification [38]. Another possibility is to analyse the information transmitted through the Inter-Component Communications (ICC) service, which can be used by malware to perform malicious actions such as installing a new application [29].

Within the set of features under a static analysis approach, *taint analysis* has also served for designing detection and classification tools. FlowDroid [42] can be considered as the most important exponent of this kind of analysis. RevealDroid [35] is an accurate and obfuscation resilient Android malware detection tool where information flows extracted with FlowDroid play a fundamental role to detect malicious patterns. Another example of the use of taint analysis is Apposcopy [16].

All the features described so far are mainly extracted through the use of static analysis techniques, what is to say, they do not imply code execution. While static features are easy to extract and provide detailed information, they present some inherent shortcomings. One of the most important examples is the inability to trace the actual behaviour of a sample when running in a real device, which prevents, in many cases, to reveal the malicious payload of a malware instance. There is an important number of researches trying to bypass this problem by monitoring the application actions in an emulator. One example is Andromaly [15], which applies machine learning to varied metrics of the system behaviour collected in runtime. Another instance of this kind of analysis is DroidScribe [26], which monitors API calls invoked in runtime to detect Android malware.

2.2. Malware detection tools and classification techniques

It is possible to find in the literature several tools aimed at detecting Android malware, as well as classifying and categorising them among families that share certain features.

For instance, RevealDroid [35] is both a malware classification and detection tool which employs a varied set of static features including sensitive API calls, API packages, API flows obtained with FlowDroid [42] and a list of actions of the Intents. Then, a C4.5 decision-tree and a 1-Nearest-Neighbour (1NN) machine learning algorithms are trained, taking as input all these features extracted from Android benign and malware samples belonging to different families. DroidSIFT [40] is a semantic-based classifier, which uses API dependency graphs to train a Naïve Bayes classifier. In Dendroid [17], code structures serve to compare samples and to train both a classification and a clustering algorithm. Drebin [18] follows the same pattern, it extracts a large of features including hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API calls and network addresses to train Support Vector Machines.

DroidMiner [25] builds behaviour graphs called Component Behaviour Graphs (CBG) based on the communication between API functions and sensitive Android resources and different classification algorithms such as SVMs or Random Forest are trained. DroidAPIMiner [20] extracts API calls based on their presence in malicious samples and are used to perform, as in previous examples, a training process of different machine learning algorithms. DroidLegacy [23] performs family classification by extracting signatures that can be found in malicious applications and that can be used in repackaged, originally benign, samples. Then, similarity measures are used to compare samples. MAST [33] helps in the triaging process making use of features extracted from the application package. Finally, MOCDROID [43] uses an evolutionary approach and third party calls to perform malware detection.

From the classification model perspective, a broad range of techniques have been used in this scenario. Among these, ensemble based classifiers have been repeatedly employed showing a good behaviour. These methods have proved to be powerful in imbalanced problems [44] as well as in binary and multi-class domains [45]. For instance, statically extracted features (including permissions, API calls and a set of system events) are used to feed a rotation forest model [21], which improves the results when compared against a classic Support Vector Machine classifier. A blend of API calls and permissions also defines the set of features employed to train a multi-feature collaborative decision fusion (MCDF) [36], through a pool of classifiers including J48, Random Tree and Decision Stump.

In DroidFusion [22], the classification is performed at two different levels. First, several low level classifiers are trained, involving Random Tree, REPTree, J48, Voted Perceptron and also ensemble methods. Then, their results are combined to define the final output using four proposed ranking-based methods. In a similar approach [37], 11 different static features categories are filtered using a SVM classifier and then are sent to the input of an ensemble classifier composed of five models: SVM, Random Forest, K-Nearest Neighbors, Classification and Regression Tree (CART) and Naive Bayes. These models are combined through a majority voting scheme.

All previous approaches, no matter the algorithm employed, require datasets of features extracted from malware and harmless samples in order to train their models.

2.3. Android malware datasets

Machine learning based malware detection tools require datasets of samples labelled as malware or benign to be trained and tested. The use of these datasets is essential in order to build reliable malware detection or classification methods. For that purpose, different datasets of Android samples have been made public over the past years. The Android Malware Genome Project [46] was launched in 2012, however, it was abandoned in 2015. Drebin [18] contains malware samples labelled as different malware families, but it is too old, since the samples were gathered from August 2010 to October 2012. Contagio [47] periodically publishes new malware applications found in the wild. The AndroZoo project [48] offers a huge dataset containing more than 5,700,000 samples which are currently being analysed with different antivirus engines. Other source where it is possible to find Android malware families is the *android-malware* GitHub repository [49] which includes APKs of different varieties. More recently, the Android Malware Dataset [50] was released. It contains 24,553 samples gathered from 2010 to 2016 of 71 malware families.

In addition to datasets, there are also online services that make it possible to retrieve both benign and malicious applications. For instance, many researchers collect samples directly from application stores, such as Google Play or Aptoide, specially when searching for benignware. On the other hand, malware can be obtained from online applications such as the VirusTotal Intelligence service[2].

While these datasets offer sets of raw samples, other projects provide features or logs already extracted from samples. In this line, DroidCat [51, 52] is composed of 440 and 508 logs of malicious and benign samples, respectively. AndroMalShare [53] also provides a dataset of features extracted from malware samples, including static and dynamic analysis. If compared to *OmniDroid*, AndroMalshare only includes malware samples and the features set is not as broad. The Kharon project [54] provides a reduced set of Android malware samples of different families and includes some technical implementation details. The Koodous portal[3] offers a huge dataset of malicious and benign samples only for research purposes, in this case including a report of features. In contrast, *OmniDroid* provides a wider set of features. Androzoo [48] provides a large set of Android Apks for research purposes.

In this work, samples provided by Koodous and Androzoo were used as the starting point in order to analyse a large set of samples considered as malware and considered as harmless.

2.4. Android malware analysis tools

To day there exist a plethora of available tools for the analysis of suspicious Android applications aimed at extracting different types of characteristics. For instance, a GitHub repository [55] offers an overview of the existing tools and links to useful resources. In particular, there is an important number of utilities focused on reverse engineering processes able to extract groups of features. For example, AndroGuard [56] is a Python library which extracts varied information from code, resources or the Android Manifest.

Other well-known tool is smali/backsmali [57], an assembler and disassembler of the DEX files which contain the bytecode of the application. Apktool [58] allows to decode the resources contained in the executable file. It also provides other powerful utilities such as repacking the sample. Dex2jar [59] is aimed at converting Dalvik bytecode files into Java compiled files with extension *jar* extension. This allows to later decompile the code using other tools such as jd-gui [60], which reconstructs the code and allows to visualise it. A similar functionality is provided by jadx [61]. FlowDroid [42] performs taint analysis over Android applications, providing useful and detailed information related to the different information flow which occur during the application lifecycle.

From the dynamic analysis perspective, DroidBox [62] enables to monitor a wide series of events such as accesses to files, network traffic or DEX files dynamically loaded in run time. An alternative to DroidBox is CuckooDroid [63]. The greatest weakness of both tools lies in that they are implemented for old versions of the Android platform.

3. AndroPyTool

Currently, there is a plethora of tools for the extraction of static and dynamic features from Android applications. However, each one focuses on a precise kind of features, so obtaining a comprehensive set of both static and dynamic features becomes an Herculean task. It involves setting up every tool with their respective configuration files and environment, as well as grouping the results obtained individually to build a complete data set. That is the motivation behind the development of AndroPyTool, in other words, to offer an all-in-one solution to the Android malware research community capable of performing a full feature extraction process from suspicious samples.

AndroPyTool is an integrated framework developed in Python aimed at obtaining varied dynamic and static features from a set of Android applications. It embeds the most used Android malware analysis tools, performs inspection on the source code and retrieves behaviour information when the sample is run within a controlled environment. The tool provides a detailed report for every analysed application, including a large batch of fine-grained representative characteristics.

Any interested researcher can get involved in the project by making improvements or fixing bugs. The source code of the project is hosted at GitHub [10]. Furthermore, AndroPyTool can be used through a Docker container, which allows to easily run the tool in just two steps. Refer to the source code repository to get a full description of the arguments and to know how to install and launch the tool without using the Docker image (i.e. using the source code).

This section comprises a description of the aforementioned tool, detailing what are the computed features.

3.1. Tool operation

As an integrated framework, AndroPyTool includes several Android analysis and processing tools to provide

[2] `https://www.virustotal.com/intelligence`

[3] `https://koodous.com/`

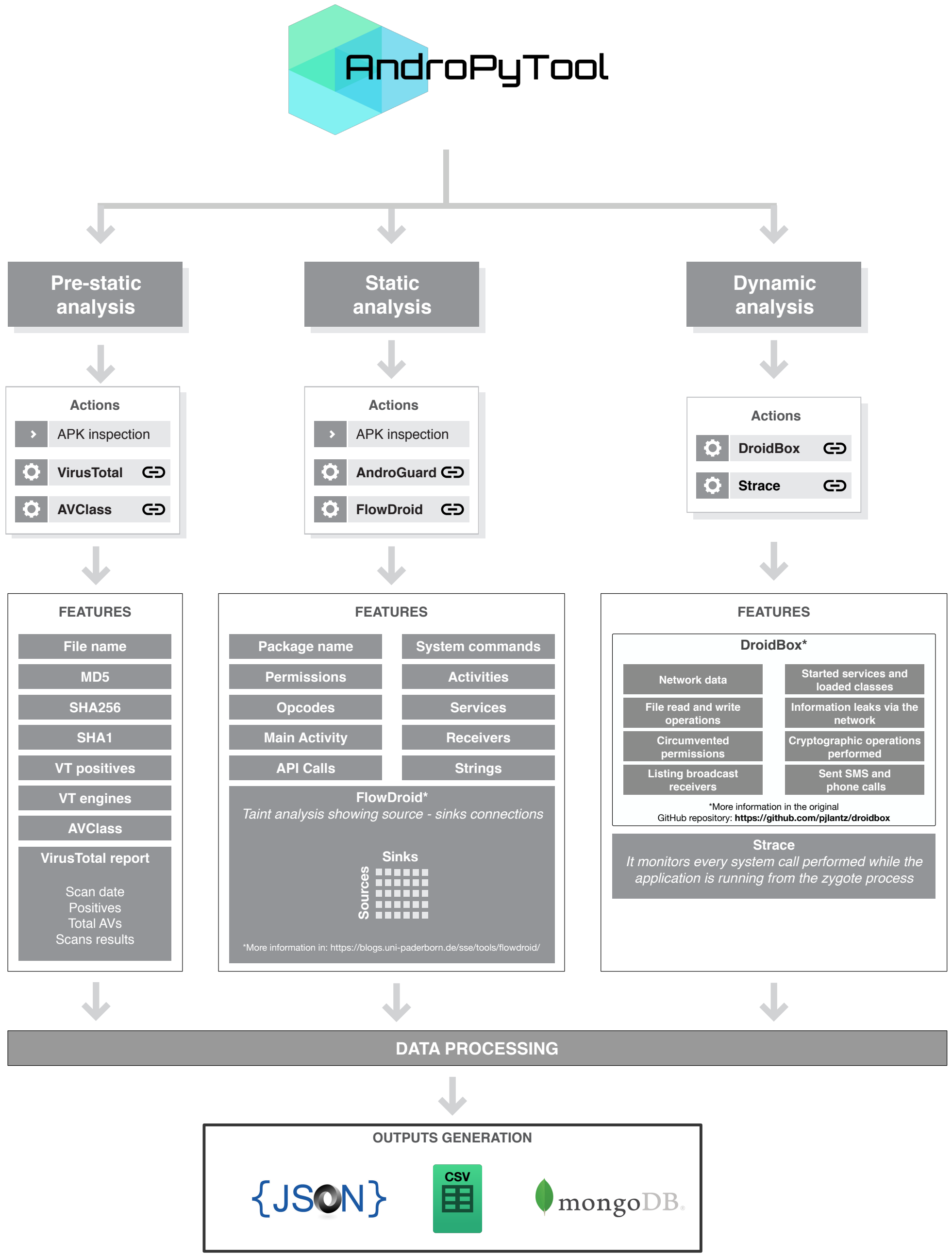


Figure 1: AndroPyTool feature extraction process. The application follows a seven steps process in which static and dynamic analysis tools are executed (steps 1–6), to finally process and group these results into an unified dataset (step 7).

detailed reports on the features and behaviour of applications. To achieve this, every application file (namely *apk*) follows a pipeline comprising seven steps (see Figure 1), which are detailed below:

1. **APK filtering**: The first step aims at inspecting every sample by using the AndroGuard tool [56] to determine whether the sample is a valid Android application.
2. **Virustotal analysis**: the tool retrieves a report of the application from the Virustotal online web application. The report contains the results from the scan performed by Virustotal as well as resulting data from the analysis of the application by more than 60 distinct anti-malware engines.
3. **Dataset partitioning**: in this step, which is optional, each sample is labelled as malware if tagged this way by at least ϵ antivirus based on the VirusTotal report. This criteria can be modified by the final user of the tool, who can establish his/her own threshold ϵ.
4. **FlowDroid execution**: this tool, based on taint analysis, is run against every sample.
5. **FlowDroid results processing**: there is a processing step of those results provided by FlowDroid,

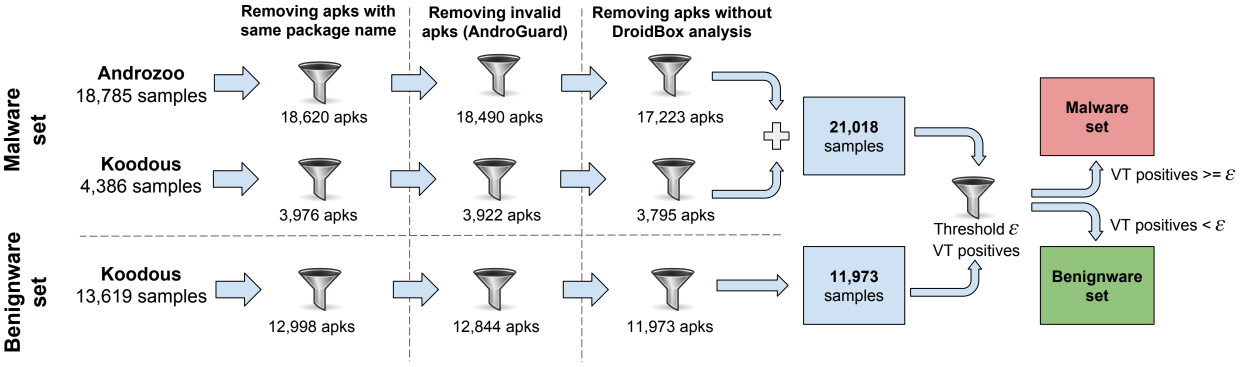


Figure 2: Filtering processes applied to build the OmniDroid dataset. The final allocation of benignware and malware samples depends on the ϵ parameter, which defines the minimum number of antivirus which test positive for malware in order to consider a sample as malicious or benign.

in order to extract the connections between sources and sinks. The reason of dividing the extraction and processing of information flows lies in that different representations are possible, thus allowing to modify the processing step independently.

6. **DroidBox execution**: a customised version of DroidBox [62], an Android dynamic analysis tool, that includes the *Strace* tool is run against the sample in this step.
7. **Feature extraction**: in this step, the tool gathers, reorganise and structure the results as reports of extracted features. The combined dataset of features extracted for all applications is provided in the following formats: as a comma-separated value (CSV), as a JavaScript Object Notation file (JSON), and as a MongoDB database.

Next section provides a description of the Omnidroid dataset, including the features extracted by the tool for every given Android application along the aforementioned process.

4. Dataset description

The OmniDroid dataset was built using AndroPyTool [10]. With this tool, a large set of samples from two different sources were analysed. In the first place, both benign and malicious samples were gathered from a dataset of 100,000 samples that the Koodous[4] Team kindly gave us for research purposes. Additional samples from the AndroZoo[5] portal have been included to supplement the first set and also to promote variety within the malware set of samples, thus avoiding potential sources of bias (virus creators, creation date, etc.).

A filtering process was followed in order to remove repeated applications and those that are considered as invalid (they cannot be actually installed and executed). As it can be seen in Fig. 2, a first filter consists of removing those samples that are represented by repeated packages names, thus avoiding several instances of the same application and their related feature vector. Then, invalid applications were detected with the AndroGuard tool [56] included into AndroPyTool in order to remove apps that cannot be executed. The third step pursues the same objective, but in this case discarding samples that could not be actually executed in the Android emulator used by DroidBox. All samples define a minimum SDK version under API 16 in their Android Manifest, ensuring that this is not a disadvantage for the use of DroidBox, which uses this API level.

AndroPyTool run on the different sets of samples until a considerable number of samples was analysed: 21,018 malware samples and 11,973 benign samples. Although the Koodous dataset already makes a distinction between malicious and non-malicious samples, all samples (including those obtained from AndroZoo) were submitted to VirusTotal. This allows to obtain an updated report for every application which includes the scan result obtained from a series of antivirus engines and which can be used to label each sample as malware or benignware. Thus, the rate of positives delivered by the antivirus engines implemented by the VirusTotal portal is included as pre-static information for each sample. Since a low rate of antivirus reporting malicious content could be due to false positives, the final allocation of samples to a malware or benignware set will be in hands of the users of the OmniDroid dataset, who can establish their own criteria. In this line, a procedure such as the one implemented by AndroPyTool can be used, where the label of each sample is set according to a threshold ϵ.

Due to the high computational load required to obtain the full analysis from each sample, the finally built dataset contains a subset of the applications gathered. This reduction was addressed aiming to keep a balanced dataset containing both malware and benignware samples. For that purpose, the threshold parameter ϵ was set to 1. According to this criteria, the finally generated and published

[4] https://koodous.com/
[5] https://androzoo.uni.lu/

dataset is composed by 11,000 malware and 11,000 benignware samples.

All the features present in this dataset have been extracted from the aforementioned set of applications by executing AndroPyTool. A general overview of the number of features extracted can be depicted in Table 2, where the most important groups of characteristics are represented. Table 1 compares the features contained in the OmniDroid dataset against the characteristics used by different methods proposed for Android malware detection and classification.

Feature	Count	Feature	Count
Permissions	5,501	Services	4,365
Opcodes	224	Receivers	6,415
API calls	2,129	API Packages	212
System commands	103	FlowDroid	961
Activities	6,089		

Table 2: Number of the most important features included in the OmniDroid dataset by category.

The next subsections describe each of the features that are provided for each sample.

4.1. Features extracted through pre-static analysis

This entry provides general information about the application, such as its file name, MD5, SHA-1 and SHA-256 checksums, a field reporting the number of positives in its VirusTotal report and the total number of antivirus engines from which scan results were obtained. Although most of these elements cannot be actually considered as behavioural features, these fields can be used to keep track of the APK for any further analysis (see Table 3). This kind of features also includes a categorisation of the sample according to the *AVClass* [54] tool, which aims to achieve a consensus between the outputs given by the different antivirus engines run by VirusTotal (see Section 4.4). This tool starts from a set of labelled samples to extract a list of tokens, detects alias of the same family, applies several filters and reveals the most convenient token for each particular sample.

4.2. Features extracted through static analysis

Once the pre-static features are obtained, a set of new features, which are extracted using static analysis techniques (see Table 4), is gathered from the samples and incorporated into the dataset. These features are: package name, permissions, opcodes, main activity's name, API calls, strings, system commands, and a list of intents whose activities, services and receivers are able to manage independently. All applications have been analysed with the *Flowdroid* [42] tool as well, so these reports are included into *OmniDroid*.

The main goal of this set of features is to provide an insight of the application expected behaviour and the range of actions that it could take based on a static analysis of the code which does not imply code execution. While this kind of analysis cannot reflect the real behaviour of the sample, which will only be revealed if the sample is run, it feeds the analysis report of a sample with valuable information. For instance, a list of permissions required offers a general picture of the range of actions that the sample could take. Furthermore, this information can also be used to compare declared and expected (static analysis) behaviour of a sample with the actually performed (dynamic analysis).

In the first place, a complete list of API calls found in the code is provided. It is important to note that those calls that are invoked through reflection or dynamic code loading among other obfuscation techniques are not detected by this kind of analysis. In the OmniDroid dataset, API calls can be found grouped by the class in which they are defined or by their package[6]. Secondly, permis-

Feature	Description
`Filename`	Filename of the APK
`VT_positives`	Number of antivirus which test positive for malware
`VT_engines`	Number of antivirus used in the analysis
`AVClass`	Agreed malware label from several detection engines according to [54]
`md5`	MD5 checksum of the APK
`sha1`	SHA-1 checksum of the APK
`sha256`	SHA-256 checksum of the APK

Table 3: Pre-static features available in the dataset.

Feature	Description
`API calls`	Count of system calls performed by an APK
`Main activity`	Name of the Main Activity
`Opcodes`	Count of opcodes performed by an APK
`Package name`	Name of the package
`Permissions`	Which permissions uses the APK
`Intent receivers`	Set of an APK's receivers
`Intent services`	Services used by an application
`Intent activities`	Activities declared by an APK
`Strings`	Set of defined strings (with use count) within an APK
`System commands`	Set of system commands ran by the app
`FlowDroid`	Path to the results obtained by FlowDroid [42]

Table 4: Static analysis features available in the dataset.

[6] A list of all Android API packages can be found at `https://developer.android.com/reference/packages.html`

sions show device functionalities that the application can use, and system commands offer an overview of the actions that can occur at a low level, so they can reveal interesting actions performed, such as privilege escalation.

Other static features include Dalvik opcodes, obtained by analysing the Dalvik bytecode and that are useful to discern the behaviour of the sample at a low level. At the same time, a list of intents that are used to invoke other application components allows to make a profile of the sample based on actions performed and the events that trigger these actions. Another kind of feature included are strings which are found within the code and that could contain sections of code prepared to be executed in run time.

Finally, a report built using the FlowDroid tool is also included in this section. This is a useful tool that performs taint analysis over the application code, in order to discover connections between a source and a sink. These sources and sinks, which have been previously defined by the SuSi framework [64], allow to model those data leaks performed during the application life-cycle. For instance, this allows to discover connections where the IMEI of the device is sent to a third-party using the network. FlowDroid was run limiting the RAM memory to 10GB and the execution time to 5 minutes.

4.3. Features extracted through dynamic analysis

This category of features groups the dynamic analysis results obtained with a customised version of the well known framework *Droidbox* [62]. This framework provides interesting features that can be analysed, such as network activity, accessed files, sent SMS and cryptographic functions that were captured during the execution. This framework[7] was adapted to obtain more detailed dynamic reports. This modification consist on deploying the Strace tool inside the Android emulator device, which allow to obtain a fine-grained list of all system calls performed in run time at the Linux level. When the analysis ends, this file is extracted and saved. Furthermore, there was also two more modifications applied to DroidBox. On the one side, the behaviour of the *MonkeyRunner* tool already implemented in DroidBox, was changed with the aim of stimulating the sample under analysis with a higher number of simulated user actions on the screen and buttons. On the other hand, this modified version allows to run the sample in a non GUI environment, thus enabling to run multiple simultaneous emulator instances in computing nodes.

In contrast to static analysis, the use of a dynamic analysis tool allows to model the real behaviour exhibited in a simulated environment where the application is executed. Giving capabilities to the sample, such as internet access, allows to capture those actions that are only visible when particular conditions are met. For instance, monitoring the suspicious application while it is being executed allows to reveal system calls which are invoked due to the use of reflection or dynamic code loading.

The OmniDroid dataset contains the whole execution log with all the events captured by the DroidBox tool and a log generated with Strace for each application analysed. These two information sources offer large amounts of data which in case of being used in combination with machine learning classifier need to be filtered and processed. The final user of this dataset is free to apply the most appropriate techniques and methods for this purpose. Furthermore, since these two logs report a sorted list of events where each one has its timestamp attached, it is also possible apply online learning techniques [65].

Each execution was run for 300 seconds in an emulator running Android 4.1.1 (the version employed by DroidBox) with an armeabi-v7a architecture.

4.4. VirusTotal report

In addition to the aforementioned features, all APKs have been analysed with the online malware detection tool VirusTotal. In fact, what this platform does is to analyse the APK using diverse malware detection engines such as: *AVG*, *Avast* and *F-Secure*, to name a few. Hence, data regarding this category are the results obtained from each malware detection engine, namely a negative/positive detection as well as the categorisation given by each antivirus engine. This report is included as a ground truth to categorise each simple sample between malicious or non malignant and to allocate a family or variant tag to each sample as well. The final user of OmniDroid is in charge of defining the ϵ threshold as the minimum number of positives which determine the nature of the sample.

4.5. Protection against adversarial attacks

Recent research has proven that adversarial attacks against machine learning aided detection tools can provoke misclassifications [66]. The wide set of features contained in the OmniDroid dataset, extracted using both a static and a dynamic analysis approach allow to build resilient detection and classification tools. While methods relying on a reduced set of features can be more easily deceived (i.e. a modification of a single feature can lead to a misclassification), those that employ a large set of characteristics to describe the application behaviour present a significant barrier against this kind of attacks.

5. Dataset analysis and benchmark

All features extracted in this dataset define a large space from which different comparisons and considerations can be made. For instance, it is possible to analyse the presence of particular features when observing malware or benign samples. For these experiments, the threshold ϵ has been set to 1, so benign samples are those that are considered as such by all antivirus engines, while those that

[7] `https://github.com/alexMyG/DroidBox_AndroPyTool`

Malware		Benignware	
Permission	**% samples**	**Permission**	**% samples**
INTERNET	96.21%	INTERNET	94.82%
ACCESS_NETWORK_STATE	85.45%	ACCESS_NETWORK_STATE	72.95%
WRITE_EXTERNAL_STORAGE	81.31%	WRITE_EXTERNAL_STORAGE	61.49%
READ_PHONE_STATE	80.21%	WAKE_LOCK	41.60%
ACCESS_WIFI_STATE	60.40%	ACCESS_WIFI_STATE	39.14%
WAKE_LOCK	49.05%	READ_PHONE_STATE	37.13%
ACCESS_COARSE_LOCATION	41.99%	VIBRATE	33.33%
GET_TASKS	39.12%	ACCESS_FINE_LOCATION	27.70%
ACCESS_FINE_LOCATION	37.00%	ACCESS_COARSE_LOCATION	27.45%
VIBRATE	36.87%	GET_ACCOUNTS	26.82%

Table 5: Ten most frequent permissions declared in the Android Manifest for each application in the malware and benignware sets.

are labelled as malware by at least antivirus are allocated to the malware set.

5.1. Dataset analysis

An evaluation of how different samples employ the different features is a useful mechanism to draw general conclusions regarding the differences between both types of samples. In the next subsection, different features are assessed individually.

5.1.1. Permissions required analysis

Table 5 shows the top 10 permissions used among the malware and the benignware set (in terms of percentage of samples where the permission is reflected in the Android Manifest). Most of the samples of both categories required Internet access in order to be executed, so no conclusion can be drawn from this fact. However, there is one which makes a big difference, the `READ_PHONE_STATE` permission. It allows to read relevant information such as the phone number, ongoing calls or cellular network information. The official Android documentation already warns about the danger of this permission, as it can be used to access very sensitive information. While about 80% of the malware malware samples require this permission, this figure decreases to 37% in the benignware set.

In general, malware is more likely to demand permissions, thus accessing a large set of functionalities. For instance, SMS related permissions are much more present among malicious samples. In particular, `SEND_SMS` or `RECEIVE_SMS` permissions are required by 29% and 24% of the malware samples respectively, while this numbers decrease to 5% within the benign set. This leads to conclude that the use of SMS services is an important indicator to consider whether a suspicious sample is malicious or not. Among the rest of permissions, the use of `RECEIVE_BOOT_COMPLETE` is also relevant in the case of malware. Most of those samples categorised as malware employ this permission to activate the malicious payload once the device has been restarted.

5.1.2. Opcodes analysis

A study on the use of opcodes has also been performed. Table 6 shows the top 10 opcodes found among the smali code in the malware and benignware sets. In general, an analysis based on counting the use of particular *opcodes* will not lead to any conclusive assessment. Proof of this, is that the top 50 opcodes are used by at least 90% of the samples, which means that no differences can be found when comparing both batches of samples. This is an expected circumstance, given that opcodes show instructions at a very high level, so they cannot be used to distinguish relevant behaviours. For instance, the use of *invoke* opcodes allows to know when a system call is invoked from the application, but without the specific call specification, no possible intentions can be inferred.

Malware		Benignware	
Opcode	**% samples**	**Opcode**	**% samples**
return-void	100.0%	return-void	100.0%
invoke-direct	100.0%	invoke-direct	100.0%
invoke-static	99.95%	invoke-super	99.99%
invoke-virtual	99.95%	invoke-virtual	99.99%
new-instance	99.95%	move-result-object	99.97%
move-result-object	99.95%	invoke-static	99.96%
const/4	99.94%	new-instance	99.96%
move-result	99.92%	goto	99.88%
goto	99.88%	move-result	99.87%
if-eqz	99.78%	const/4	99.87%

Table 6: Ten most frequent opcodes found in the smali code obtained for each application in the malware and benignware sets.

5.1.3. System commands analysis

Malware		Benignware	
System command	**% samples**	**System command**	**% samples**
id	67.14%	id	68.56%
start	53.38%	top	66.83%
gzip	41.52%	start	59.55%
date	39.55%	service	59.04%
service	32.61%	gzip	52.35%
log	32.48%	input	47.41%
input	27.59%	mv	46.85%
stop	26.65%	log	43.41%
sh	25.91%	sh	42.89%
top	24.98%	stop	37.33%

Table 7: Ten most frequent system commands found within the code of each application in the malware and benignware sets.

Contrary to the trends seen in the usage of permissions by both types of samples, in the *OmniDroid* dataset the

Malware		Benignware	
API Package	**% samples**	**API Package**	**% samples**
android.app	99.96%	android.app	99.65%
java.lang	99.91%	java.lang	99.56%
java.io	98.8%	android.content	99.44%
android.content	97.85%	java.io	99.02%
android.os	94.58%	android.content.res	98.57%
android.content.res	94.0%	android.widget	96.65%
android.util	91.45%	android.os	96.56%
android.net	91.42%	android.view	96.47%
java.util	90.25%	java.lang.ref	96.47%
android.widget	90.05%	android.util	96.28%

Table 8: Ten most frequent API packages referenced on each application in both the malware and benignware sets.

use of system commands were more significant among benign samples (see Table 7). However, there can be found important details among them. For instance, 19% of the malware samples make use of the `chmod` command, which is necessary to give write and execution permissions to hidden scripts included among the app files. Among the benign samples, this number is reduced to 5%. In contrast, a wide set of system commands that could be tentatively associated with malicious behaviours, are in fact more used among benign samples. For instance, the commands *gzip* and *mv* could be used to decompress a file containing a malicious payload and to place it in the system apps folder, however, both are more commonly employed among benign samples. The rest of system commands do not allow to extract any relevant conclusion that could make a difference between both kinds of samples.

5.1.4. API packages analysis

API calls form a useful mechanism to identify relevant differences between samples with good or non-legal intentions. In this study, these calls are grouped by the API packages in which they are defined, in order to depict general characteristic patterns. In table 8 it is possible to observe this ranking of number of calls per API package. In general, both types of samples exhibit a similar use of the most common API packages. For instance, the *android.app* and *java.lang* packages, which are present in all samples, include the most basic functionality of the Android environment and Java language features respectively. An important difference can be found in the *android.telephony* groups of API calls. While 83% of the malicious samples invoke calls contained in this package, this number is reduced to 63% in the benignware set. As it was already observed when analysing the use of different permissions, services related to telephony and SMS remain very present among malware samples.

5.1.5. Intents analysis

Intents are a key communication element in the Android environment. Basically, they allow to ask permission to the system to run another application component. Thus, Intents describe the actions that an application can take. For instance, an Intent could demand actions such as making a phone call or taking a picture. Table 9 shows the top 10 actions declared in the Android Manifest by Intent-filters grouped by the nature of the samples. There are noticeable patterns which can be inferred from this ranking. Permission *android.intent.action.*`BOOT_COMPLETED` is declared in the Android Manifest in order to allow the application to receive a broadcast intent reporting when the device has been restarted. This is typically employed by malware aiming to hide the malicious payload until the next device boot-up and to force that a malicious application is actually executed even if the user does not start it manually. While about 38% of the malware samples are listening to this broadcast, only 19% of the benign apps do it. Another examples of broadly used permissions among malicious samples is *android.intent.action.*`USER_PRESENT` or those related to the SMS services (as already noted when analysing the use of Android permissions) such as *android.provider.Telephony.*`SMS_RECEIVED`.

5.2. FlowDroid analysis

Information flows represent vital information to understand the behaviour of a sample and are able to reveal patterns that could be associated to a sequence of malicious actions. In order to assess the differences between the malware and the benignware sets, Fig. 3 shows the number of flows found across all samples between categories of sources (at the left) and sinks (at the right). These 31 categories are defined by the SuSi framework [64].

As it can be seen, there is an important number of links between the `NOT_EXISTING` and `NO_CATEGORY` (those related to non-private data) categories. These categories have been omitted in the rest of the study. When analysing information flows in the malware diagram starting from the `DATABASE_INFORMATION` category, an important number reach the `LOCATION_INFORMATION` category, mainly focused on getting information regarding the device location.

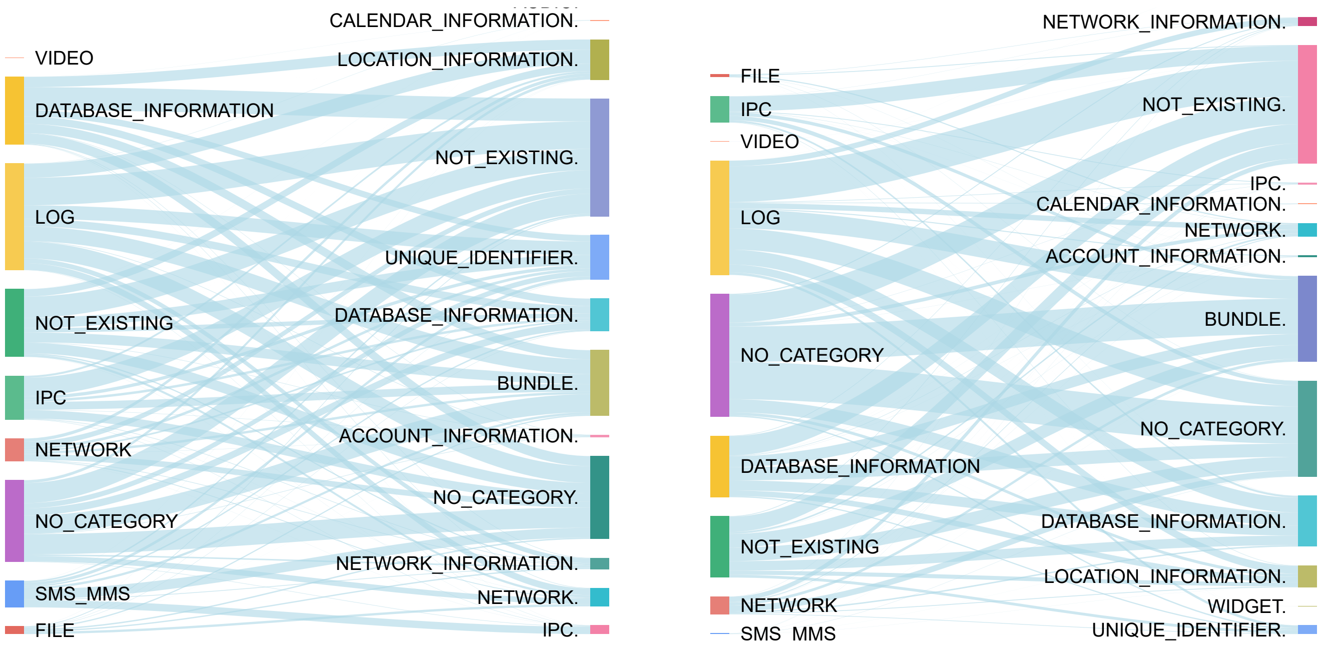
There are also important patterns which can be deduced from the malware set. For example, there is a big number of information flows from `SMS_MMS` to `IPC` (interprocess communication). In contrast, `SMS_MMS` category shows little activity in the benign set. At the same time, there is more activity in this set starting from `IPC` and `FILE` categories. As sink, `UNIQUE_IDENTIFIER` category receives an important number of flows, which is a remarkable fact, since it includes relevant calls such as `getDeviceId()` or `getImei()`. These calls are often used by malware for many purposes. For instance, they can be used to generate unique keys which are later employed to encrypt user's files. Another important category with a similar behaviour in both datasets is `LOG`, which includes a broad range of actions such as those related to wallpaper or policies managing.

5.3. Dynamic features analysis

The dynamic traces obtained with *DroidBox* have also been studied. In general terms, both collections show

Malware		Benignware	
Intent	**% samples**	**Intent**	**% samples**
android.intent.action.MAIN	99.86%	android.intent.action.MAIN	99.81%
android.intent.action.BOOT_COMPLETED	38.05%	android.intent.action.VIEW	21.64%
android.net.conn.CONNECTIVITY_CHANGE	26.01%	com.google.android.c2dm.intent.RECEIVE	21.45%
android.intent.action.USER_PRESENT	25.80%	android.intent.action.BOOT_COMPLETED	19.46%
android.intent.action.PACKAGE_ADDED	24.00%	com.google.android.c2dm.intent.REGISTRATION	14.74%
android.intent.action.VIEW	16.96%	android.net.conn.CONNECTIVITY_CHANGE	11.65%
android.provider.Telephony.SMS_RECEIVED	15.42%	com.android.vending.INSTALL_REFERRER	10.67%
android.intent.action.PACKAGE_REMOVED	15.15%	android.intent.action.USER_PRESENT	6.73%
com.google.android.c2dm.intent.RECEIVE	11.47%	android.intent.action.PACKAGE_REMOVED	5.95%
com.google.android.c2dm.intent.REGISTRATION	8.78%	android.appwidget.action.APPWIDGET_UPDATE	5.68%

Table 9: Ten most frequent Android intents in the apps from both malware and benignware sets.



(a) (b)

Figure 3: Number of information flows between categories of sinks and sources discovered by FlowDroid among all samples in the (a) malware set and in the (b) benignware set.

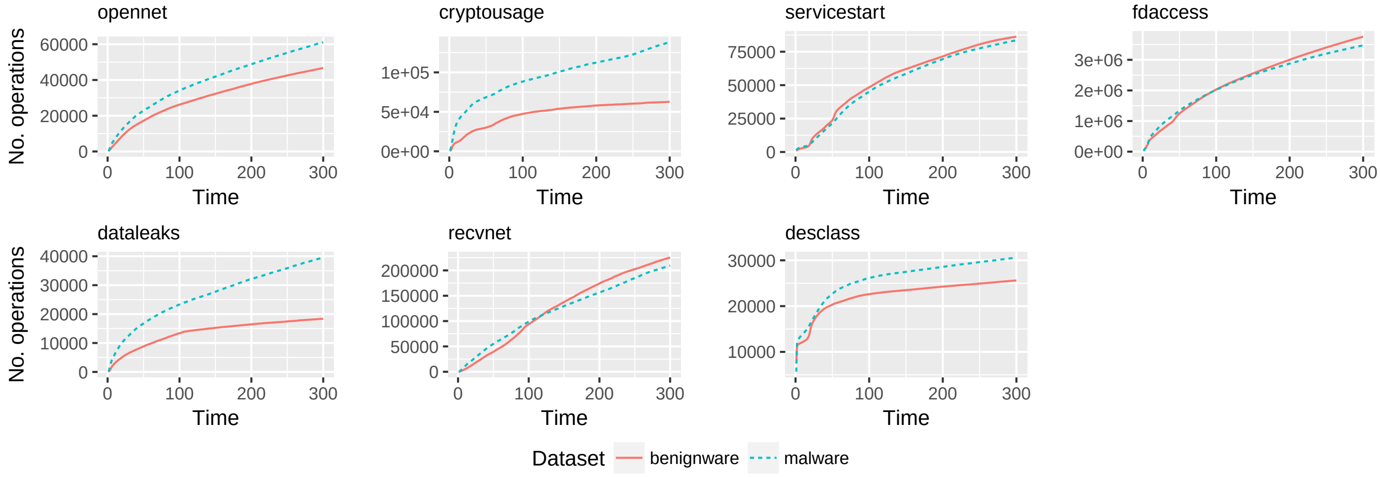


Figure 4: Cumulative sum of the number of operations, when *Droidbox* tool is used, for all samples over both (benignware and malware) datasets.

Dynamic feature	Malware	Benignware
accessedfiles	315.74	341.44
cryptousage	12.56	5.70
dataleaks	3.59	1.67
dexclass	2.78	2.33
fdaccess	315.72	341.42
opennet	5.56	4.25
phonecalls	0.01	0.02
recvnet	19.05	20.48
sendnet	4.85	2.65
sendsms	0.21	0.00
servicestart	7.60	7.84

Table 10: Average number of dynamic operations per execution in the malware and benignware set.

a very similar behaviour when they are executed (see Table 10). The most relevant differentiating factor lies in the number of cryptographic operations performed by the malware set. On average, malicious samples invoke 12.56 operations, while in the benignware set this figure falls by half. A greater use of SMS services can also be reflected in these dynamic traces, which are more present in the malware set.

Fig. 4 shows the cumulative sum of the number of operations through the 300 seconds execution in the emulator for each different category of operations. In general, malware deploys a slightly more active behaviour, a fact which is more remarkable in the case of the `cryptousage` and `dataleaks` categories. In the rest of categories, both sets perform very close. For instance, `servicestart` and `fdaccess` show a completely parallel behaviour. Finally, it should be noted that both kinds of samples mainly load code from *dex* classes in the first 30 seconds.

6. Testing the data using ensemble-based classification algorithms

Many malware detection and classification tools are based on machine learning algorithms, performing a learning process from a training set of samples represented by a set of features. The purpose of the process described in this section is to demonstrate the feasibility and ease of use of the dataset when using it to build classifiers through machine learning methods (in particular, ensemble methods), rather than building a malware classifier achieving a high accuracy. In other words, show that the dataset is usable out-of-the-box, and although some promising experimental results are currently obtained, there is still a large room for improvement. These methods were used separately over the set of static and dynamic features extracted, and finally a fusion based approach where both types of features are combined is proposed. The same threshold related to the minimum number of positives applied in the previous section, defined by $\epsilon = 1$, is again used to train and test these algorithms.

6.1. Classification results based on static features

Statically extracted features can be used to build representative sample vectors where each position represents the number of occurrences that a certain characteristic is present in the sample (i.e. the number of times that a specific API call is invoked). Given a set of samples X of size n:

$$X = \{x_1, x_2, \dots, x_n\} \tag{1}$$

Each sample x_i is represented by a vector of m static characteristics:

$$x_i = \{sc_i^1, sc_i^2, \dots, sc_i^m\} \tag{2}$$

At the same time, each sample x_i is categorised as benign or malicious according to its label l_i, $l_i \in \{0, 1\}$ Our objective is to train the classifier than establishes this relation:

$$Cls(x_i) = (p(l_i), l_i) \tag{3}$$

Six well-known state-of-the-art ensemble methods for classification were trained with different combinations of features. These algorithms, executed with the *Scikit-learn* library for Python [67], are: AdaBoost, Bagging (with Random Forest estimators), ExtraTrees, Gradient Boosting, Random Forest (all of them using a parameter of 100 internal estimators) and a Voting classifier combining a Random Forest, KNN and a simple decision tree classifier as estimators with the same weights. The *CSV* file used in these experiments (containing all the labelled features vectors) is also publicly available.

For all the experiments, based on different features combinations, the average accuracy is shown. The results for each experiment is calculated based on a cross validation process of 10 folds. Table 11 depicts the classification results for individuals features as the only input for the classification and for different combinations of features as well.

By analysing the results achieved by each static feature, API calls allow to obtain the maximum accuracy with a random forest classifier, reaching 89.3%. If grouping these API calls by the API package in which they are defined, these features become useless to distinguish between malware and benignware. In the second place, appears the combinations that include the use of API calls, achieving around 89.2% accuracy, which can be considered as a high and also similar value. Other features such as *FlowDroid*, a feature which apparently could reveal important traces related to malicious behaviours (such as sending the device IMEI to an attacker) or system commands are not able to make a division of the feature space with the same level of precision. In general, random forest clearly achieves the best results in most of the experiments in terms of accuracy and precision. Nevertheless, a Bagging classifier obtains similar results for certain combinations of features.

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Activities	**Acc**	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002	0.506 ± 0.002
	Prec	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036	0.602 ± 0.036
API calls	**Acc**	0.859 ± 0.008	**0.891 ± 0.007**	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.886 ± 0.006
	Prec	0.859 ± 0.008	**0.892 ± 0.007**	0.89 ± 0.006	0.871 ± 0.009	**0.893 ± 0.006**	0.887 ± 0.006
API Packages	**Acc**	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0	0.5 ± 0
	Prec	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0	0.25 ± 0
FlowDroid	**Acc**	0.677 ± 0.009	0.706 ± 0.008	0.708 ± 0.008	0.681 ± 0.01	0.708 ± 0.008	0.704 ± 0.009
	Prec	0.72 ± 0.013	0.744 ± 0.009	0.748 ± 0.009	0.723 ± 0.013	0.746 ± 0.009	0.744 ± 0.01
FlowDroid, API calls	**Acc**	0.86 ± 0.01	**0.891 ± 0.007**	0.889 ± 0.006	0.872 ± 0.007	**0.892 ± 0.007**	0.886 ± 0.006
	Prec	0.86 ± 0.01	**0.892 ± 0.007**	0.89 ± 0.006	0.872 ± 0.007	**0.892 ± 0.007**	0.886 ± 0.006
FlowDroid, API Packages	**Acc**	0.677 ± 0.009	0.708 ± 0.008	0.709 ± 0.008	0.681 ± 0.01	0.707 ± 0.01	0.704 ± 0.008
	Prec	0.72 ± 0.013	0.745 ± 0.009	0.749 ± 0.009	0.722 ± 0.013	0.745 ± 0.011	0.745 ± 0.01
Opcodes	**Acc**	0.833 ± 0.011	0.873 ± 0.01	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.012	0.868 ± 0.009
	Prec	0.833 ± 0.011	0.873 ± 0.009	0.869 ± 0.007	0.846 ± 0.009	0.874 ± 0.011	0.868 ± 0.009
Permissions	**Acc**	0.781 ± 0.01	0.824 ± 0.006	0.824 ± 0.006	0.792 ± 0.008	0.825 ± 0.007	0.821 ± 0.008
	Prec	0.781 ± 0.01	0.826 ± 0.006	0.826 ± 0.006	0.792 ± 0.008	0.827 ± 0.006	0.823 ± 0.008
Receivers	**Acc**	0.824 ± 0.005	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
	Prec	0.825 ± 0.006	0.876 ± 0.01	0.877 ± 0.009	0.84 ± 0.005	0.877 ± 0.01	0.875 ± 0.009
Receivers, API calls	**Acc**	0.858 ± 0.01	0.889 ± 0.006	0.89 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
	Prec	0.858 ± 0.01	0.889 ± 0.006	0.891 ± 0.008	0.875 ± 0.007	**0.892 ± 0.008**	0.885 ± 0.007
Receivers, API calls,	**Acc**	0.862 ± 0.009	0.89 ± 0.008	0.891 ± 0.008	0.88 ± 0.008	**0.891 ± 0.007**	0.884 ± 0.008
Opcodes, Permissions	**Prec**	0.862 ± 0.009	0.89 ± 0.008	0.891 ± 0.008	0.88 ± 0.008	**0.892 ± 0.007**	0.884 ± 0.008
Receivers, API calls, Opcodes,	**Acc**	0.865 ± 0.008	0.889 ± 0.007	0.892 ± 0.009	0.879 ± 0.008	**0.891 ± 0.008**	0.883 ± 0.007
Permissions, FlowDroid	**Prec**	0.865 ± 0.008	0.89 ± 0.007	0.893 ± 0.009	0.88 ± 0.008	**0.892 ± 0.008**	0.883 ± 0.007
Receivers, Services,	**Acc**	0.825 ± 0.005	0.875 ± 0.008	0.877 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
Activities	**Prec**	0.826 ± 0.005	0.875 ± 0.008	0.878 ± 0.007	0.843 ± 0.008	0.876 ± 0.008	0.876 ± 0.008
Receivers, Services,	**Acc**	0.858 ± 0.01	0.889 ± 0.007	0.888 ± 0.006	0.874 ± 0.008	0.889 ± 0.007	0.884 ± 0.007
Activities, API calls	**Prec**	0.858 ± 0.01	0.889 ± 0.007	0.889 ± 0.006	0.874 ± 0.008	0.89 ± 0.007	0.884 ± 0.007
Services	**Acc**	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.002	0.515 ± 0.003	0.516 ± 0.002	0.516 ± 0.003
	Prec	0.749 ± 0.013	0.741 ± 0.015	0.743 ± 0.015	0.75 ± 0.012	0.741 ± 0.015	0.742 ± 0.015
System commands	**Acc**	0.761 ± 0.009	0.827 ± 0.007	0.827 ± 0.007	0.776 ± 0.007	0.826 ± 0.006	0.82 ± 0.008
	Prec	0.763 ± 0.009	0.828 ± 0.007	0.828 ± 0.007	0.777 ± 0.007	0.827 ± 0.006	0.821 ± 0.008

Table 11: Performance of several ensemble learning algorithms from the state of the art according to the static feature set used as input. The best overall results are highlighted in bold type.

Features set	Metric	AdaBoost	Bagging	ExtraTrees	Gradient Boosting	Random Forest	Voting
Transitions	**Acc**	0.731 ± 0.01	0.776 ± 0.01	0.775 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.763 ± 0.009
Transitions	**Prec**	0.731 ± 0.01	0.777 ± 0.01	0.776 ± 0.012	0.741 ± 0.007	0.775 ± 0.009	0.764 ± 0.009
Frequencies	**Acc**	0.739 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.009	0.778 ± 0.011	0.768 ± 0.008
Frequencies	**Prec**	0.74 ± 0.009	0.78 ± 0.012	0.774 ± 0.01	0.743 ± 0.008	0.778 ± 0.011	0.769 ± 0.008
Combination	**Acc**	0.742 ± 0.009	**0.786 ± 0.007**	0.779 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.771 ± 0.011
Combination	**Prec**	0.743 ± 0.009	**0.786 ± 0.008**	0.78 ± 0.007	0.751 ± 0.006	**0.785 ± 0.006**	0.772 ± 0.011

Table 12: Performance of several ensemble learning algorithms from the state of the art according to the dynamic feature set used as input. The best overall performance is highlighted in bold type.

6.2. Classification results based on dynamic features

Once the static features have been analysed, it is tested the use of the dynamic information, extracted after the execution of each sample in an emulator monitored by the DroidBox tool. Each analysis delivers a temporal sequence of actions performed throughout the execution, where each action is linked to a category (i.e. file access), a timestamp and a series of parameters (i.e. the path of the file accessed). In order to build a feature vector which can be used to represent the sample behaviour, a Markov chains based representation [68] was employed, as previously were used by Martín et al. [69] to model DroiBox dynamic traces. This model allows to represent the transitions probabilities a_{ij} between a series of n states:

$$S = \{S_1, S_2, \ldots, S_n\} \tag{4}$$

The full dynamic trace of a certain sample x_i is firstly transformed into a $n * n$ matrix of n unique states, which represent the transition probability between each pair of states. Each state is represented by the category of the action and a series of arguments. An example of state is: `fdaccess\operation=read|path=/proc/tty`, which indicates a file operation of type *read* over a file located at `/proc/tty`. All paths were truncated to limit their depth to two levels. In addition, only those transitions with $a_i > 0$ were considered in order to reduce the matrix size. A final set of 1,127 states were obtained.

In order to place this information into a feature vector, the matrix of each sample was flattened, thus generating a representative vector of k transitions probabilities:

$$x_i = \{tp_i^1, tp_i^2, \ldots, tp_i^k\} \tag{5}$$

At the same time, the frequency of each state were also considered, as it can provide supplementary data able to improve the representation of the sample behaviour. In this case, there is a new representative vector including the frequency of o states for each sample:

$$x_i = \{sf_i^1, sf_i^2, \ldots, sf_i^o\} \tag{6}$$

Both representations can be combined in order to build a new vector containing transition probabilities and frequencies of states:

$$x_i = \{tp_i^1, tp_i^2, \ldots, tp_i^k, sf_i^1, sf_i^2, \ldots, sf_i^m\} \tag{7}$$

These representations were implemented an tested with the same pool of algorithms tested previously with the static features vectors. The results obtained are shown in Table 12, and include the classification accuracy and precision using both representations independently, and a third one combining transitions and states frequencies. Again, a 10-fold cross validation is employed.

Opposite of what one would expect, the use of frequencies states allows to reach higher accuracy values than using the transition probabilities among states, meaning that the distribution of states provides a better description of the samples behaviour. However, a combination of both features allows to achieve the best results, by using a Bagging model composed of Random Forest classifiers or a single Random Forest. Both allow to obtain 78.6% in terms of accuracy and precision.

In comparison to the results achieved through the use of static features, dynamically extracted characteristics experience an important fall of more than 10%. There are several reasons which can be attributed to this fact. On the one side, reports delivered by the dynamic analysis tool used, DroidBox, could be not enough detailed in order to make a division of the space able to differentiate between malicious and benign applications. Besides, DroidBox has not been updated in the last years and could be not able to monitor the most recent malware applications behaviours. However, there are other plausible explanations. For instance, malware samples have proven to be able to detect when they run in a sandbox, thus not deploying the malicious payload [70]. For the classifier perspective, there is space for future users of the OmniDroid dataset for applying techniques aimed at building stronger estimators. For instance, the use of diversity-inducing methods can help in this task [71].

6.3. Static-dynamic fusion method for detecting Android malware

The two previous sections describe how the state-of-the-art ensemble methods works when our set of static and dynamic features are given as a feature vectors for classification. As it has been shown, the use of a classifier with an input based on static features allows to improve the accuracy when it is compared against a dynamic based approach. Despite of this, the growing complexity of malware makes necessary to apply any available techniques, to discern the nature of a given suspicious sample. To this end, a new approach was developed based on fusion of the behaviour information extracted from a hybrid analysis where the static and dynamic features are combined.

This fusion approach is based on joining the classification models that work best in each type of feature (static and dynamic) to build a voting classifier where each model contributes to the final categorisation. It features three different representations to generate the classifier. While it adopts API calls as static features (they obtain the best results in the static comparison), it also uses the three representations analysed in the dynamic analysis approach. Thus, there is a mixture of API calls features which are combined with transition probabilities (equation 8), frequencies states (equation 9) and combination of both (equation 10) to build three vectors:

$$xtp_i = \{sc_i^1, sc_i^2, \ldots, sc_i^m, tp_i^1, tp_i^2, \ldots, tp_i^k\} \tag{8}$$

$$xsf_i = \{sc_i^1, sc_i^2, \ldots, sc_i^m, sf_i^1, sf_i^2, \ldots, sf_i^o\} \tag{9}$$

$$\begin{aligned} xdc_i = \{&sc_i^1, sc_i^2, \ldots, sc_i^m, tp_i^1, tp_i^2, \\ &\ldots, tp_i^k, sf_i^1, sf_i^2, \ldots, sf_i^o\} \end{aligned} \tag{10}$$

The design of the voting classifier implemented is represented in Fig. 5. It includes a Random Forest classifier which is in charge of classifying the static features section of the input vector, while a Bagging classifier (it slightly exceed the results of Random Forest) receives dynamic features. Each classifier contributes to the classification of a given feature according to two weight parameters W_{RF} and W_{BG}. The final categorisation is calculated as follows:

$$Cls(x_i) = \left[\tfrac{W_{RF}S_i + W_{BG}D_i}{2}\right] = l_i, l_i \in \{0, 1\} \tag{11}$$

A grid search was run to decide the best value for W_{RF} and W_{BG} within the range $[0.1, 0.9]$. The results are shown in Table 13. As it could be expected, and according to the results previously seen, when analysing the use dynamic features as the only input, in these experiments the best results are also achieved by combining transition probabilities and frequencies of states. The best result is achieved using $W_{RF} = 0.7$ and $W_{BG} = 0.3$, after a 10-fold cross validation process. This fusion approach achieves 89.7% accuracy, which slightly improves the results when both types of features are used independently.

7. Conclusions

Android conforms a platform that has been selected as the target by many black hats to perform malicious

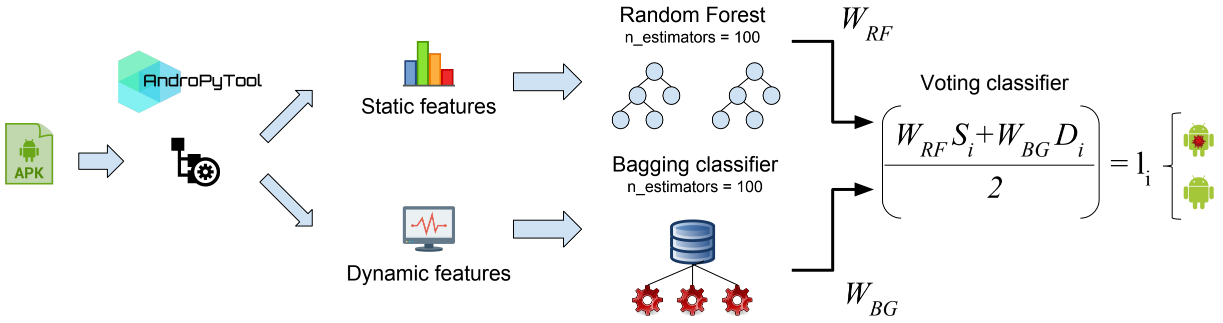


Figure 5: Static-dynamic fusion approach based on a voting classifier which combines static and dynamic classification models.

$\mathbf{W_{RF}}$	$\mathbf{W_{BG}}$	**Metric**	**Transitions**	**Frequencies**	**Combination**
0.1	**0.9**	**Acc**	0.812 ± 0.01	0.81 ± 0.01	0.815 ± 0.011
		Prec	0.813 ± 0.01	0.81 ± 0.01	0.816 ± 0.011
0.2	**0.8**	**Acc**	0.834 ± 0.01	0.83 ± 0.007	0.837 ± 0.009
		Prec	0.835 ± 0.01	0.83 ± 0.007	0.838 ± 0.009
0.3	**0.7**	**Acc**	0.86 ± 0.008	0.854 ± 0.008	0.861 ± 0.007
		Prec	0.86 ± 0.008	0.855 ± 0.008	0.862 ± 0.007
0.4	**0.6**	**Acc**	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
		Prec	0.879 ± 0.006	0.873 ± 0.007	0.88 ± 0.007
0.5	**0.5**	**Acc**	0.891 ± 0.007	0.887 ± 0.008	0.892 ± 0.007
		Prec	0.891 ± 0.006	0.887 ± 0.007	0.892 ± 0.007
0.6	**0.4**	**Acc**	0.896 ± 0.01	0.895 ± 0.008	0.894 ± 0.01
		Prec	0.896 ± 0.01	0.896 ± 0.008	0.894 ± 0.01
0.7	**0.3**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.008**
		Prec	0.895 ± 0.008	0.896 ± 0.008	**0.897 ± 0.007**
0.8	**0.2**	**Acc**	0.895 ± 0.008	0.896 ± 0.008	0.895 ± 0.007
		Prec	0.895 ± 0.008	0.896 ± 0.008	0.896 ± 0.007
0.9	**0.1**	**Acc**	0.893 ± 0.008	0.893 ± 0.008	0.894 ± 0.006
		Prec	0.893 ± 0.008	0.894 ± 0.008	0.894 ± 0.006

Table 13: Results achieved with the static-dynamic fusion approach. W1 and W2 represent the weight given to the Random Forest classifier and to the Bagging classifier in the voting-based approach, responsible of receiving as input the static and dynamic features respectively.

attacks, or to develop applications with non-legal purposes. Many of the efforts made towards counteracting these attacks are based on the training process of an antimalware tool. This process, specially when is based on a machine learning algorithm, requires from a large dataset of samples to be adequately trained. Previously to use this kind of algorithms, it is necessary to extract and analyse an adequate set of features that could be used during the learning process. *OmniDroid* primarily aims to set a benchmark dataset useful for those who are developing antimalware tools. Instead of providing a set of executable files, this dataset provides already analysed samples using different state-of-the-art malware analysis tools in order to facilitate the process. The characteristics of the *OmniDroid* dataset make it also suitable to be used as a general purpose benchmark dataset. For instance, it could be employed to perform algorithms comparison, to test feature selection techniques, or to assess new clustering or classification algorithms among many others possibilities.

Throughout this paper, we have evaluated this dataset, performed different studies and provided results after analysing groups of features individually, in order to deliver interested researchers with a preliminary evaluation of the data, as well as demonstrating the high potential the dataset has and its ease of use. Although the experimental results show a good performance for most of the state-of-the art algorithms analysed, there is still a clear room for improvement over OmniDroid dataset. This could help to a researcher to train, test and evaluate the performance of their algorithms, or simply to compare their malware detection techniques using OmniDroid as a new, clean and correctly pre-processed, data benchmark.

Finally, we are considering to expand the architecture of *AndroPyTool* [10], in order to allow extracting more features for each sample by integrating other feature extraction and reverse engineering tools. At the same time, we will work in the near future with the goal of increasing the number of samples contained in the *OmniDroid* dataset, updating it to introduce recent samples found in the wild. The OmniDroid dataset, and their future updates, will be available for the research community at the *AIDA Datasets Repository*[8].

Acknowledgement

This work has been co-funded by the following grants: Comunidad Autónoma de Madrid under grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks); Spanish Ministry of Science and Education and Competitivity (MINECO) and European Regional Development Fund (FEDER) under grants TIN2014-56494-C4-4-P (EphemeCH), and TIN2017-85727-C4-3-P (DeepBio). We also would like to thank to the Koodous[9] Team for the large dataset of samples that they provided us.

References

[1] Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.M.. DDoS in the IoT: Mirai and other botnets. IEEE Computer 2017;50(7):80–84.

[2] Ehrenfeld, J.M.. Wannacry, cybersecurity and health information technology: A time to act. Journal of Medical Systems 2017;41(7):104:1.

[8] https://aida.ii.uam.es/datasets/

[9] `https://koodous.com/`

[3] IDC, . Worldwide Quarterly Mobile Phone Tracker. Tech. Rep.; International Data Corporation (IDC); Massachusetts, USA; 2017.

[4] Chandrasekar, K., Cleary, G., Cox, C., Lau, H., Nahorney, B., O'Gorman, B., et al. Internet security threat report. Tech. Rep.; Symantec Corporation; California, USA; 2017.

[5] Quinlan, J.R.. C4.5: programs for machine learning. Elsevier; 2014.

[6] Boser, B.E., Guyon, I., Vapnik, V.. A training algorithm for optimal margin classifiers. In: Haussler, D., editor. Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992. ACM; 1992, p. 144–152.

[7] Langley, P., Iba, W., Thompson, K.. An analysis of bayesian classifiers. In: Swartout, W.R., editor. 10th National Conference on Artificial Intelligence. AAAI Press / The MIT Press; 1992, p. 223–228.

[8] Dietterich, T.G.. Ensemble methods in machine learning. In: Multiple Classifier Systems, First International Workshop, MCS 2000; vol. 1857 of *Lecture Notes in Computer Science*. Springer; 2000, p. 1–15.

[9] Martín, A., Lara-Cabrera, R., Camacho, D.. A new tool for static and dynamic android malware analysis. In: Data Science and Knowledge Engineering for Sensing Decision Support. World Scientific; 2018, p. 509–516.

[10] AndroPyTool source code repository. 2017. URL: `https://github.com/alexMyG/AndroPyTool`; Last accessed: 2018-05-04.

[11] Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0). 2013. URL: `https://creativecommons.org/licenses/by-nc-sa/4.0/`; Last accessed: 2018-03-14.

[12] Martín, A., Calleja, A., Menéndez, H.D., Tapiador, J.E., Camacho, D.. ADROIT: android malware detection using meta-information. In: 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016. IEEE; 2016, p. 1–8.

[13] Jang, J., Kang, H., Woo, J., Mohaisen, A., Kim, H.K.. Andro-dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information. Computers & Security 2016;58:125–138.

[14] Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.. Androdialysis: Analysis of android intent effectiveness in malware detection. computers & security 2017;65:121–134.

[15] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weiss, Y.. "Andromaly": a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems 2012;38(1):161–190.

[16] Feng, Y., Anand, S., Dillig, I., Aiken, A.. Apposcopy: semantics-based detection of android malware through static analysis. In: Cheung, S., Orso, A., Storey, M.D., editors. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22). ACM; 2014, p. 576–587.

[17] Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Blasco, J.. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. Expert Systems with Applications 2014;41(4):1104–1117.

[18] 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. The Internet Society; 2014, p. 1–12.

[19] 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. IEEE Computer Society; 2013, p. 163–171.

[20] Aafer, Y., Du, W., Yin, H.. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: 9th International ICST Conference, SecureComm 2013; vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer; 2013, p. 86–103.

[21] Zhu, H.J., You, Z.H., Zhu, Z.X., Shi, W.L., Chen, X., Cheng, L.. Droiddet: Effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing 2018;272:638–646.

[22] Yerima, S.Y., Sezer, S.. Droidfusion: A novel multilevel classifier fusion approach for android malware detection. IEEE Transactions on Cybernetics 2018;doi:`10.1109/TCYB.2017.2777960`.

[23] Deshotels, L., Notani, V., Lakhotia, A.. Droidlegacy: Automated familial classification of android malware. In: Jagannathan, S., Sewell, P., editors. 3rd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW 2014. ACM; 2014, p. 3:1–3:12.

[24] 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. IEEE; 2012, p. 62–69.

[25] Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P.A.. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: 19th European Symposium on Research in Computer Security, ESORICS 2014; vol. 8712 of *Lecture Notes in Computer Science*. Springer; 2014, p. 163–182.

[26] Dash, S.K., Suarez-Tangil, G., Khan, S.J., Tam, K., Ahmadi, M., Kinder, J., et al. DroidScribe: classifying android malware based on runtime behavior. In: 2016 IEEE Security and Privacy Workshops, SP Workshops 2016. IEEE Computer Society; 2016, p. 252–261.

[27] Suarez-Tangil, G., Dash, S.K., Ahmadi, M., Kinder, J., Giacinto, G., Cavallaro, L.. DroidSieve: fast and accurate classification of obfuscated android malware. In: Ahn, G., Pretschner, A., Ghinita, G., editors. Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017. ACM; 2017, p. 309–320.

[28] Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.. Structural detection of android malware using embedded call graphs. In: Sadeghi, A., Nelson, B., Dimitrakakis, C., Shi, E., editors. 2013 ACM Workshop on Artificial Intelligence and Security, AISec'13. ACM; 2013, p. 45–54.

[29] Xu, K., Li, Y., Deng, R.H.. ICCDetector: ICC-based malware detection on Android. IEEE Transactions on Information Forensics and Security 2016;11(6):1252–1264.

[30] Dini, G., Martinelli, F., Saracino, A., Sgandurra, D.. Madam: a multi-level anomaly detector for android malware. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. Springer; 2012, p. 240–253.

[31] Feldman, S., Stadther, D., Wang, B.. Manilyzer: Automated android malware detection through manifest analysis. In: 11th IEEE International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2014. IEEE Computer Society; 2014, p. 767–772.

[32] Lindorfer, M., Neugschwandtner, M., Platzer, C.. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In: Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual; vol. 2. IEEE; 2015, p. 422–433.

[33] Chakradeo, S., Reaves, B., Traynor, P., Enck, W.. MAST: triage for market-scale mobile malware analysis. In: Buttyán, L., Sadeghi, A., Gruteser, M., editors. Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC'13. ACM; 2013, p. 13–24.

[34] Peiravian, N., Zhu, X.. Machine learning for android malware detection using permission and API calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. IEEE Computer Society; 2013, p. 300–305.

[35] Garcia, J., Hammad, M., Pedrood, B., Bagheri-Khaligh, A., Malek, S.. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Tech. Rep.; George Mason University; Virginia, USA; 2015.

[36] Sheen, S., Anitha, R., Natarajan, V.. Android based malware detection using a multifeature collaborative decision fusion

approach. Neurocomputing 2015;151:905–912.

[37] Wang, W., Li, Y., Wang, X., Liu, J., Zhang, X.. Detecting android malicious apps and categorizing benign apps with ensemble of classifiers. Future Generation Computer Systems 2018;78:987–994.

[38] Yerima, S.Y., Sezer, S., McWilliams, G., Muttik, I.. A new android malware detection approach using bayesian classification. In: Barolli, L., Xhafa, F., Takizawa, M., Enokido, T., Hsu, H., editors. 27th IEEE International Conference on Advanced Information Networking and Applications, AINA 2013. IEEE Computer Society; 2013, p. 121–128.

[39] Yerima, S.Y., Sezer, S., Muttik, I.. Android malware detection using parallel machine learning classifiers. In: Eighth International Conference on Next Generation Mobile Apps, Services and Technologies, NGMAST 2014. IEEE; 2014, p. 37–42.

[40] Zhang, M., Duan, Y., Yin, H., Zhao, Z.. Semantics-aware android malware classification using weighted contextual API dependency graphs. In: Ahn, G., Yung, M., Li, N., editors. 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. ISBN 978-1-4503-2957-6; 2014, p. 1105–1116.

[41] Canfora, G., Lorenzo, A.D., Medvet, E., Mercaldo, F., Visaggio, C.A.. Effectiveness of Opcode ngrams for detection of multi family android malware. In: 10th International Conference on Availability, Reliability and Security, ARES 2015. IEEE Computer Society; 2015, p. 333–340.

[42] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: O'Boyle, M.F.P., Pingali, K., editors. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM; 2014, p. 259–269.

[43] Martín, A., Menéndez, H.D., Camacho, D.. MOCDroid: multi-objective evolutionary classifier for Android malware detection. Soft Computing 2017;21(24):7405–7415.

[44] Galar, M., Fernandez, A., Barrenechea, E., Bustince, H., Herrera, F.. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 2012;42(4):463–484.

[45] Galar, M., Fernández, A., Barrenechea, E., Bustince, H., Herrera, F.. An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes. Pattern Recognition 2011;44(8):1761–1776.

[46] Zhou, Y., Jiang, X.. Dissecting android malware: Characterization and evolution. In: IEEE Symposium on Security and Privacy, SP 2012. IEEE Computer Society; 2012, p. 95–109.

[47] Contagio malware dumps. 2008. URL: `https://contagiodump.blogspot.com.es`; Last accessed: 2018-03-14.

[48] Allix, K., Bissyandé, T.F., Klein, J., Traon, Y.L.. Androzoo: collecting millions of android apps for the research community. In: Kim, M., Robbes, R., Bird, C., editors. 13th International Conference on Mining Software Repositories, MSR 2016. ACM; 2016, p. 468–471.

[49] Android-malware source code repository. 2016. URL: `https://github.com/ashishb/android-malware`; Last accessed: 2018-03-14.

[50] Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer; 2017, p. 252–276.

[51] Rashidi, B., Fung, C.J.. Xdroid: An android permission control using hidden markov chain and online learning. In: 2016 IEEE Conference on Communications and Network Security, CNS 2016. IEEE; 2016, p. 46–54.

[52] Rashidi, B., Fung, C.J., Bertino, E.. Android resource usage risk assessment using hidden markov model and online learning. Computers & Security 2017;65:90–107.

[53] AndroMalShare dataset. 2013. URL: `http://sanddroid.xjtu.edu.cn:8080/`; Last accessed: 2018-03-14.

[54] Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.. Avclass: A tool for massive malware labeling. In: 19th International Symposium on Research in Attacks, Intrusions, and Defenses, RAID 2016. 2016, p. 230–253.

[55] android-security-awesome. 2018. URL: `https://github.com/ashishb/android-security-awesome`; Last accessed: 2018-11-06.

[56] Androguard. 2011. URL: `https://github.com/androguard/androguard`; Last accessed: 2018-11-06.

[57] smali/backsmali. 2018. URL: `https://github.com/JesusFreke/smali`; Last accessed: 2018-11-06.

[58] Apktool. 2018. URL: `https://ibotpeaches.github.io/Apktool/`; Last accessed: 2018-11-06.

[59] dex2jar. 2018. URL: `https://github.com/pxb1988/dex2jar`; Last accessed: 2018-11-06.

[60] jd-gui. 2018. URL: `https://github.com/java-decompiler/jd-gui`; Last accessed: 2018-11-06.

[61] jadx. 2018. URL: `https://github.com/skylot/jadx`; Last accessed: 2018-11-06.

[62] Droidbox source code repository. 2011. URL: `https://github.com/pjlantz/droidbox`; Last accessed: 2018-03-14.

[63] Cuckoo-droid. 2018. URL: `https://github.com/idanr1986/cuckoo-droid`; Last accessed: 2018-11-06.

[64] Rasthofer, S., Arzt, S., Bodden, E.. A machine-learning approach for classifying and categorizing android sources and sinks. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014. The Internet Society; 2014, p. 1–15.

[65] Gomes, H.M., Barddal, J.P., Enembreck, F., Bifet, A.. A survey on ensemble learning for data stream classification. ACM Computing Surveys (CSUR) 2017;50(2):23.

[66] Calleja, A., Martín, A., Menéndez, H.D., Tapiador, J., Clark, D.. Picking on the family: Disrupting android malware triage by forcing misclassification. Expert Systems with Applications 2018;95:113–126.

[67] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 2011;12:2825–2830.

[68] Fink, G.A.. Markov models for pattern recognition: from theory to applications. Springer Science & Business Media; 2014.

[69] Martín, A., Rodríguez-Fernández, V., Camacho, D.. Candyman: Classifying android malware families by modelling dynamic traces with markov chains. Engineering Applications of Artificial Intelligence 2018;74:121–133.

[70] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security. ACM; 2014, p. 5.

[71] Lobo, J.L., Laña, I., Ser, J.D., Bilbao, M.N., Kasabov, N.. Evolving spiking neural networks for online learning over drifting data streams. Neural Networks 2018;108:1–19.