

Article

Two Anatomists Are Better than One—Dual-Level Android Malware Detection

Vasileios Kouliaridis ^{1,†} , Georgios Kambourakis ^{2,*,†} , Dimitris Geneiatakis ^{2,†} 
and Nektaria Potha ^{1,†}

¹ Department of Information & Communication Systems Engineering, University of Aegean, 83200 Karlovasi Samos, Greece; bkouriaridis@aegean.gr (V.K.); nekpotha@aegean.gr (N.P.)

² European Commission, Joint Research Centre (JRC), 21027 Ispra, Italy; dimitrios.geneiatakis@ec.europa.eu

* Correspondence: georgios.kampourakis@ec.europa.eu or gkamb@aegean.gr; Tel.: +39-0332-785-013

† These authors contributed equally to this work.

Received: 1 June 2020; Accepted: 29 June 2020; Published: 7 July 2020



Abstract: The openness of the Android operating system and its immense penetration into the market makes it a hot target for malware writers. This work introduces Androtomist, a novel tool capable of symmetrically applying static and dynamic analysis of applications on the Android platform. Unlike similar hybrid solutions, Androtomist capitalizes on a wealth of features stemming from static analysis along with rigorous dynamic instrumentation to dissect applications and decide if they are benign or not. The focus is on anomaly detection using machine learning, but the system is able to autonomously conduct signature-based detection as well. Furthermore, Androtomist is publicly available as open source software and can be straightforwardly installed as a web application. The application itself is dual mode, that is, fully automated for the novice user and configurable for the expert one. As a proof-of-concept, we meticulously assess the detection accuracy of Androtomist against three different popular malware datasets and a handful of machine learning classifiers. We particularly concentrate on the classification performance achieved when the results of static analysis are combined with dynamic instrumentation vis-à-vis static analysis only. Our study also introduces an ensemble approach by averaging the output of all base classification models per malware instance separately, and provides a deeper insight on the most influencing features regarding the classification process. Depending on the employed dataset, for hybrid analysis, we report notably promising to excellent results in terms of the accuracy, F1, and AUC metrics.

Keywords: mobile malware; malware analysis; machine learning classification; hybrid detection; android security

1. Introduction

Every year the number of mobile malicious applications (apps) in the wild increases significantly. For instance, according to a 2020 report by McAfee [1], hidden apps, known for their ability to conceal their presence after installation while annoying victims with invasive ads, have become the most active mobile threat. To stay covert, malware writers encrypt their code and spread it throughout the program, thus making legacy signature-based detection increasingly harder. Overall, the galloping rise of mobile malware of any kind calls for more robust detection solutions by leveraging on Machine Learning (ML).

Generally, mobile malware detection schemes can be categorized into two broad classes, namely signature and anomaly-based. The former collects patterns and signatures stemming from known malware, and compares them against unknown pieces of code for determining their status. The latter class employs a more lax approach; by observing the normal behavior of a piece of code for a certain

amount of time and using the metrics of that normal model against any deviant behavior. App analysis on the other hand is done by means of either a static, dynamic, or hybrid method. Briefly, opposite to the dynamic approach, static analysis involves examining an app without actually running it. In the last few years, considerable work has been devoted to all the three types of app analysis for the sake of effectively detecting malware on Android [2–6]. However, hitherto, little attention has been paid to hybrid approaches for malware detection with emphasis on ML.

The work at hand concerns the design, implementation and evaluation of Androtomist, a hybrid detection solution that symmetrically combines static analysis and dynamic instrumentation to detect mobile malware on the Android platform. Instrumentation is the process of injecting trace code into a software's source code, binary code, or execution environment, and is mainly exploited for debugging, tracing, and performance and security analysis purposes. Instrumentation techniques do not necessarily modify code, but rather tamper with the app's execution or behavior based on defined constructs. The advantages of melding static analysis with dynamic instrumentation into a hybrid approach are twofold. On the one hand, the static analysis works as a fast pre-filtering mechanism capitalizing on permission analysis, API calls, and intents. On the other, by instrumenting code, one is able to scrupulously analyze and detect a plethora of behaviors during runtime, such as network traffic, information leakage threatening the privacy of the end-user [7], and so on. On top of legacy signature-based metrics, Androtomist is assessed via the use of different classification performance metrics against three well-known malware datasets.

To enable the use of Androtomist by virtually anyone, including ordinary end-users, a web app has been developed. The web-based interface caters for easily uploading and analyzing mobile apps on-the-fly. Both static and dynamic analysis in terms of feature extraction are performed automatically. Finally, unlike similar open source tools, Androtomist is easy to set up and has only a few dependencies, all of which pertain to open source software. The main contributions of this study are:

- A methodology is presented that collects groups of static and dynamic features mapping the behavior of an app, including permissions, intents, API calls, Java classes, network traffic, and inter-process communication. Especially for the dynamic analysis part, among others, we improve the hitherto related work by means of contributing features emanating from the hooking of Java classes, which is made possible due to instrumentation.
- We examine the effect of features when dynamic instrumentation is provided. It is demonstrated that the effectiveness of classification models based on either static or dynamic analysis without instrumentation is outperformed by models using dynamic instrumentation.
- We report experimental results on three different well-known mobile malware benchmark datasets that are directly compared with state-of-the-art methods under the same settings. The performance of the approaches presented in this paper is quite competitive to the best results reported so far for these corpora, demonstrating that the proposed methods can be an efficient and effective alternative toward more sophisticated malware detection systems.
- We propose an ensemble approach by averaging the output of all base models for each malware instance separately. The combination of all base models achieves the best average results in all three data sets examined. This demonstrates that ensembles of classifiers based on multiple, possibly heterogeneous models, can further improve the performance of individual base classifiers.
- A new, publicly available, open source software tool is provided that could be exploited for conducting further research on mobile malware detection.

The remainder of this paper is organized in the following manner. The next section discusses the related work. Section 3 details on our methodology. Section 4 presents our results and outcomes. A discussion of the key findings is included in Section 5. The last section concludes and provides pointers to future work.

2. Related Work

As of today, the topic of mobile malware detection via the use of hybrid schemes has received significant attention in the Android security literature [8,9]. This section along with Table 1 offer a succinct chronologically arranged review of the most notable and recent works on this topic. Specifically, we concentrate on contributions published over the last four years, that is, from 2015 to 2020.

Patel et al. [10] proposed a hybrid approach for mobile malware detection, which combines app analysis and ML techniques to classify apps as benign or malicious. The features collected vary from app's permissions, intents, and API calls during static analysis to network traffic in the course of dynamic analysis. According to the authors, it takes around 8 to 11 min to scan an app.

Cam et al. [11] presented *uitHyDroid*. Their system firstly uses static analysis to collect user's interface elements, followed by dynamic analysis to capture possible inter-app communication and link partial sensitive data flows stemming from static analysis. However, their proposed system has been evaluated with a rather small dataset of 23 apps.

Tuan et al. [12] proposed *eDSDroid*, an analysis tool which uses both static and dynamic analysis to identify malware targeting inter-app communication. Specifically, at a first stage, static analysis detects information leakage, while dynamic analysis is employed to help eliminate the false positives of the first stage. On the downside, their approach has only been evaluated with a tiny corpus of 3 apps. Martinelli et al. [13] introduced *BRIDEMAID*. Their system operates in three consequent steps, namely static, meta-data, and dynamic. During static analysis, *BRIDEMAID* decompiles the apk and analyzes the source code for finding possible similarities in the executed actions. This is done with the help of *n-grams*. Dynamic analysis exploits both ML classifiers and security policies to control suspicious activities related to text messages, system call invocations, and administrator privilege abuses. In their evaluations, the authors used the Drebin dataset, which by now is mostly considered outdated.

Xu et al. [14] proposed *HADM*. With *HADM*, the features extracted during static analysis are converted into vector-based representations, while those fetched during dynamic analysis, namely system call invocations, are converted into vector-based and graph-based representations. Deep learning techniques were used to train a neural network for each of the vector sets. Finally, the hierarchical multiple kernel learning technique was applied with the purpose of combining different kernel learning results from diverse features, and thus improve the classification accuracy. According to the authors, their model is weak against code obfuscation techniques because dynamic analysis is guided based on the results obtained from static analysis.

Ali-Gombe et al. [15] presented *AspectDroid* an hybrid analysis system, which analyzes Android apps to detect unwanted or suspicious activities. The proposed system employs static bytecode instrumentation to provide efficient dataflow analysis. However, static instrumentation is unable to detect apps which use anti-unpacking and anti-repackaging obfuscation mechanisms. Furthermore, the authors used the Drebin dataset to evaluate their model, which is mostly considered obsolete nowadays. Finally, their tests were conducted on an outdated Android version (ver. 6.0).

Arshad et al. [16] introduced a hybrid malware detection scheme, namely *SAMADroid*. According to the authors, *SAMADroid* delivers high detection accuracy by combining static and dynamic analysis, which run both in a local and remote fashion. ML is used to detect malicious behavior of unknown apps and to correctly classify them. On the negative side, their scheme was tested on an outdated Android version (ver. 5.1).

Tsutano et al. [17] contributed an Android analysis framework called *JITANA*. According to the authors, *JITANA* can be used to identify inter-app communications as well as potential malicious collaborations among apps. A number of tests conducted by the authors demonstrated the effectiveness of *JITANA* in three different Android devices with multiple installed apps. On the negative side, the authors neither provide the detection rate and precision achieved nor any result about malicious apps which do not use intents.

Wang et al. [18] proposed a hybrid approach coined *DirectDroid*, which combines fuzzy logic with a new type of app testing process called "on-demand forced execution". According to the authors,

by using this technique one can trigger more hidden malicious behaviors. On top of that, DirectDroid avoids app crashes with the use of fuzzy logic and static instrumentation. DirectDroid was evaluated against 951 malware samples from 3 datasets, namely Drebin, GitHub, and AMD, and it is reported that it can detect more malicious behaviors via a method called “augmenting fuzzing”. Nevertheless, given that code obfuscation techniques render detection harder for virtually any static approach, DirectDroid cannot detect malware which employs code obfuscation or have malicious code in native payload. Put simply, any obfuscated code is invisible to DirectDroid’s static analyzer.

Fang et al. [19] suggested a hybrid analysis method which performs dynamic analysis on the results of static analysis. During the static analysis phase, they decompiled the app’s APK file to extract permissions from the manifest file as well as any occurrence of API features existing in *smali* files. Regarding dynamic analysis, they generated user input and logged the system calls. Finally, they used ML algorithms to classify apps. After experimentation, they reported a detection accuracy of 94.6% using a balanced dataset of 4000 benign and 4000 malicious apps.

Lately, Surendran et al. [20] implemented a Tree Augmented Naive Bayes (TAN) model that combines the classifier output variables pertaining to static and dynamic features, namely API calls, permissions and system calls, to detect malicious behavior. Their experiments showed an accuracy of up to 97%. The authors did not provide information about the Android version they employed or about how they generated user input events for the purposes of dynamic analysis.

Table 1. Outline of the related work.

Work	Year	Method	Dataset	Limitations
[10]	2015	Permissions, intents, API calls, network traffic. Employs ML to classify apps.	DroidKin, Contagio Mobile	Slow (8 to 10 min to analyze an app)
[11]	2017	Analyzes APK to fetch possible data flows. Analyzes UI elements. Detects inter-app communication.	DroidBench	Small dataset (23 samples)
[12]	2017	Analyzes APK to retrieve API calls and data flows. Detects inter-app communication and data leakage.	ToyApps	Small dataset (3 samples)
[13]	2017	Logs system calls, privileges, and text messages. Uses n-gram classification.	Drebin, Genome, Contagio Mobile	Outdated dataset
[14]	2018	Combines app’s features with others derived by deep learning collected during static and dynamic analysis to classify an app.	VirusShare	Unable to detect malware which uses obfuscation techniques
[15]	2018	Uses static bytecode instrumentation and dataflow analysis.	Drebin	Small dataset (100 samples) Outdated Android version (6.0)
[16]	2018	Creates feature vectors from permissions, intents, API calls, system calls, and uses ML to classify apps.	Drebin	Outdated Android version (5.1)
[17]	2019	Detects inter-app communication.	Unspecified	Metrics and results are insufficient
[18]	2019	Augments fuzzing with on-demand forced execution	Drebin, AMD, GitHub	Uses obsolete Android version (4.4) Weak against obfuscation techniques
[19]	2019	Analyzes permissions, API references, and system calls	VirusShare	Limited information is given about the examined malware samples collected
[20]	2020	Analyzes permissions, API references, and system calls and uses TAN to predict malicious behavior	Drebin, AMD AndroZoo, Github	Android version is not provided User input method is not provided

As observed from Table 1, scarce research is hitherto conducted toward the use of dynamic instrumentation in hybrid detection approaches, as none of the above mentioned contributions exploits such a scheme. Furthermore, none of the previous work capitalises on information stemming from hooking Java classes. Moreover, none of them is offered as a web app and only the work in Reference [17] is open source. ML techniques are exploited in half of the works included in the Table, namely References [10,13,14,16,19]. Finally, it is worth mentioning that there exists a number of

free online Android app analysis tools, including *AMAAaaS* [21] and *VirusTotal* [22]. *AMAAaaS* reportedly applies hybrid analysis, but unfortunately its source code is kept private, and the tool's webpage provides insufficient information regarding its analysis and classification engines. On the other hand, *VirusTotal* is not an apk analysis tool, but rather a vote-based system that leverages analysis from multiple engines.

3. Methodology

This section details on our methodology, that is, the tool implemented, the feature extraction process, and the datasets exploited.

3.1. Androtomist

Androtomist is offered as a web app [23], which enables the user to easily upload, analyze, and study the behavior of any Android app. Figure 1 depicts a high-level view of the Androtomist's architecture. Specifically, Androtomist comprises three basic components, namely virtual machines (VMs), database (DB), and a web app. Each app is installed on a VM running the Android OS. All the features collected during app analysis are stored in the DB. A user is able to interact with Androtomist via the web app.

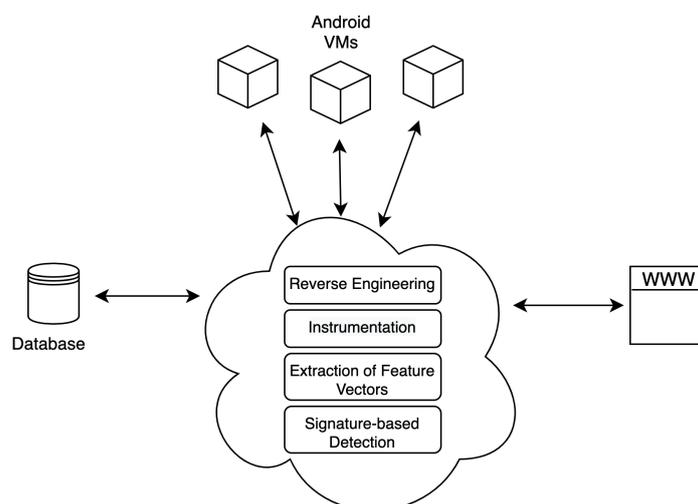


Figure 1. Androtomist's high level architecture.

For the needs of this work, Androtomist runs on a desktop computer equipped with an Intel Core i7-3770K CPU and 8GB of RAM. VirtualBox [24] is used to create and manage Virtual Machines (VMs). Depending on the user's preference, a VM running Android ver. 7.1, 8.0 or 9.0 is automatically deployed during an app's dynamic analysis phase in order to extract its runtime behavior. A VM communicates with Androtomist using ADB [25], which is a command-line tool that enables communication between an Android device and a computer, and also provides access to a Unix shell. Each time after the installation and testing of an app, the device (VM) is reverted to its original state. This guarantees the best possible comparison results and simultaneously ensures the forensic soundness of the data [26]. This process can be easily performed automatically when using VMs, and as result, the execution time needed for the whole process is substantially reduced. In the context of this work, VMs running the Android ver. 7.1 were used. The reason for not choosing a newer version is to achieve the maximum possible compatibility across malware samples from each dataset listed in Section 3.3. Finally, Androtomist is written in .NET Core and its source code is publicly available at Reference [27] under the European Union Public Licence [28].

The process of analyzing APK files is simple and does not demand any special knowledge about malware analysis. After the user connects to the Androtomist's app, they are able to upload the desired APK file in the "files" section. Next, after clicking on the corresponding button, the system initiates the

analysis process, which executes in the background. The output per APK file analyzed is available for inspection in the “results” webpage, where there is also an option to export it to a CSV-formatted file. The results webpage incorporates a summary of the app’s malicious features, if any, plus an overall percentage estimation on whether the app is malevolent or not. A detailed output report is also available for download, which contains a substantial amount of data, namely permissions, intents, network traffic, and inter-app communications. Generally, inter-app communication refers to the action of data being passed on to another app using intents during runtime. However, it is noted that inter-app communication cannot always be detected while analyzing one app at a time. Androtomist yields the following types of data:

- Data from static analysis: Permissions and intents from the manifest file, API calls retrieved from the smali files.
- Data from dynamic analysis by means of instrumentation: Network traffic, inter-app communication, Java classes.

All the above mentioned pieces of data are stored in the DB. The web app offers two types of user menu, “user” and “admin”. The latter is meant only for the expert user and offers advanced functions, such as rebuilding the detection signatures from new datasets (see Section 4.1), exporting feature vectors to train ML models (see Section 4.2), and adding users to the app. The former is destined to the novice user, even the non-tech-savvy one, and offers basic functionality, including uploading and analyzing apps, as well as inspecting the analysis results.

3.2. Extraction of Features and Feature Modeling

Under the hood, Androtomist firstly decompiles and scans the app’s APK file to extract information. Secondly, it installs and runs the APK on a VM, while generating pseudorandom user events, including clicks, touches, gestures, as well as a number of system-level events to extract the app’s runtime behavior. In addition, the tool produces vectors of features that can be fed to ML classifiers to assort apps. The internal workings of both static and dynamic analysis phases is schematically shown in Figure 2.

Precisely, the app’s APK is fed ① to the analysis controller, which first triggers static and then dynamic analysis. During static analysis ②, the APK file is scrutinised to collect static features as detailed further down. During dynamic analysis ③, a VM running the Android OS is loaded, and the APK is installed on it ④. The instrumentation code is injected during runtime ⑤. Moreover, pseudorandom user input events are generated ⑥ and the instrumentation controller logs any information emanated from the triggered events ⑦. The totality of features collected by both static and dynamic analysis are stored in the DB. The following subsections elaborate on each of the aforementioned steps of the analysis process.

Static feature extraction: The APK file is used as input to fetch static features, that is, permissions, intents, and API calls. These features are automatically extracted with the help of reverse engineering techniques. Indicatively, for a 1 Mb, 10 Mb, and 20 Mb app, the overall process takes correspondingly ≈ 10 , ≈ 16 , and ≈ 19 secs. In greater detail, the main steps are as follows:

- Use the Apktool [29] to decompile the APK file and get the manifest file called “AndroidManifest.xml” and the binary stream file called “classes.dex”.
- Get app’s permissions and intents from the manifest file.
- Decompile the binary stream to acquire the smali files and scan them to discover API calls.

Dynamic feature extraction: Androtomist collects a variety of information during app runtime by means of dynamic instrumentation. These pieces of data include the Java classes used, network communication specifics, and inter-app communication details. To do so, Androtomist capitalises on Frida [30], a full-fledged free dynamic instrumentation toolkit. Frida is able to hook on any function,

inject code into processes, and trace the app’s code without executing any compilation steps or app restarts. In particular, the instrumentation controller illustrated in Figure 2 feeds ⑦ the instrumentation code along with the app’s package name to Frida during runtime. The instrumentation code hooks on method implementations and alters their operation. Information logged by Frida is forwarded to Androtomist in real-time using ADB. As previously mentioned, all the aforementioned functions are performed automatically. The overall process takes ≈2 min for all apps, independent of their size. Specifically, 1 min is required to load a VM and install the app, and another one is allocated to the generation of ≈1000 pseudorandom user action sequences using the UI/app exerciser Monkey [31]. The main stages can be summarized as follows:

- Start a new VM with a clean image of the Android OS.
- Connect the device (VM) with ADB.
- Copy the selected APK file in the device (VM) and install it.
- Start the instrumentation process using the package name and the instrumentation code.
- Generate pseudorandom streams of user events into the app on the VM.
- Log the output from the instrumentation process, namely java classes, inter-app communication derived from triggered intents, and any information stemming from network communications, including communication protocols, sockets, IPs, ports, and URLs.

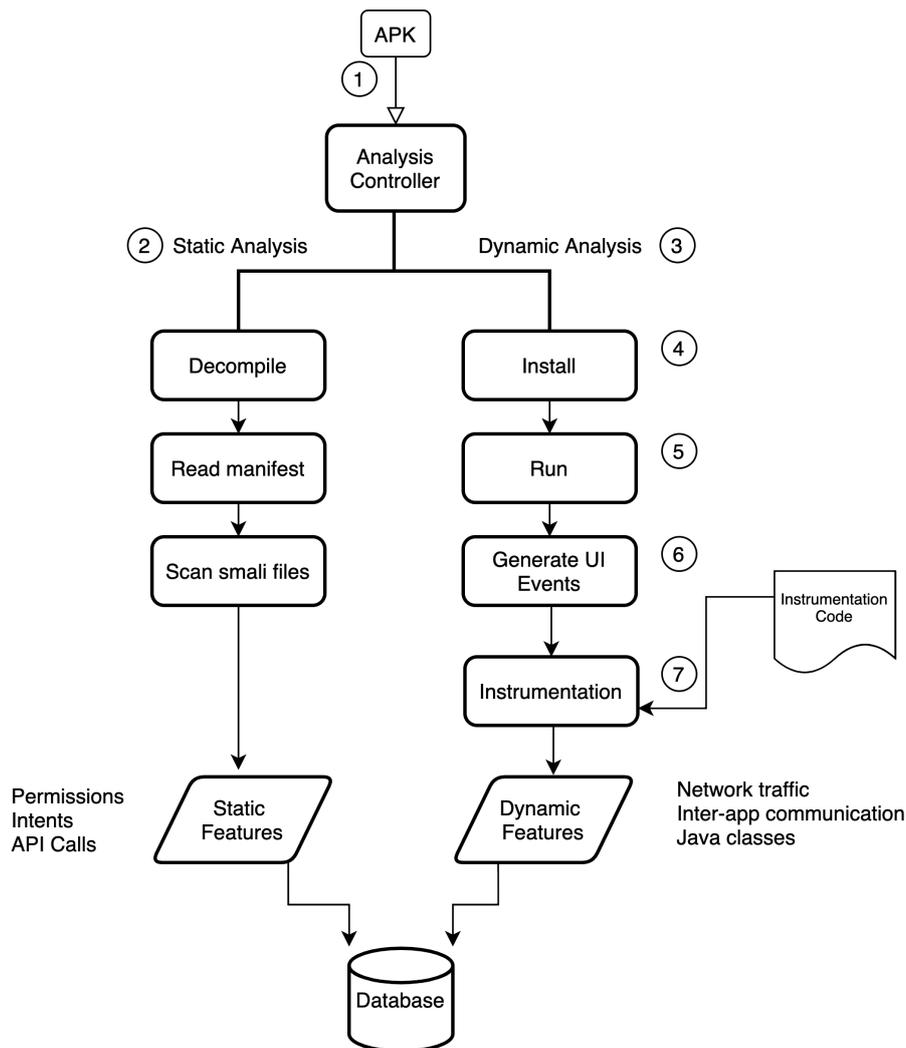


Figure 2. Androtomist’s components interworking.

Androtomist allows experienced users to create and import custom Javascript instrumentation code and inject it during dynamic analysis. The code in Listing 1 is a simplified example of such a case and can be applied to log possible intents triggered during runtime. Specifically, the code hooks the constructors of the class *android.content.Intent*, which is used by Android apps to trigger intents.

Listing 1. Example of instrumentation code.

```

1  /*The primary pieces of information in intents are:
2  1. Action
3  The general action to be performed, such as
4  ACTION_VIEW, ACTION_EDIT, ACTION_MAIN, etc.
5  2. Data
6  The data to operate on, such as a person record in the contacts DB,
7  expressed as a URI*/
8
9  Java.perform(function() {
10 var intent = Java.use("android.content.Intent");
11
12 //constructor Intent(String action)
13 intent.$init.overload('java.lang.String').implementation=function(action)
14 {
15 console.log("action" + action);
16 }
17
18 //constructor Intent(Intent o)
19 intent.$init.overload('android.content.Intent').implementation=function(o)
20 {
21 console.log("intent" + o);
22 }
23
24 //constructor Intent(String action, Uri uri)
25 intent.$init.overload('java.lang.String', 'android.net.Uri')
26 .implementation=function(action, uri)
27 {
28 console.log("action" + action + ", url" + uri);
29 }
30 ...
31 });

```

The data collected in the DB can be further analyzed to classify an app as malicious or benign. To this end, as it is illustrated in Figure 3, the gathered raw data must be first transformed into vectors of features. As already pointed out, the resulting vectors can be exported to a CSV-formatted file and used in the context of any data mining tool.

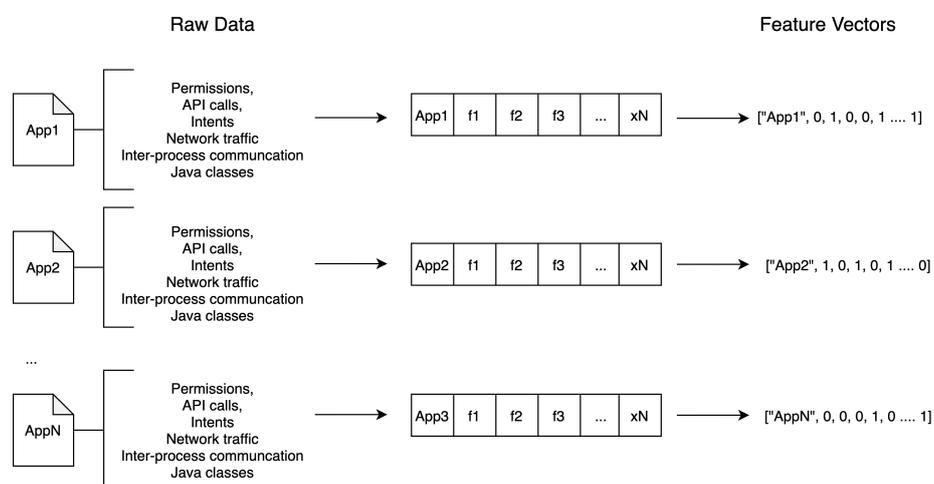


Figure 3. Feature engineering. f_i corresponds to an existing feature.

3.3. Dataset

In the context of mobile malware detection, several datasets were built to cover manifold and exceptionally demanding malicious cases. These corpora of malware samples are considered as benchmark datasets for evaluating novel schemes that attempt to distinguish between malware and goodware samples. Initially, this work employed three well-known datasets used extensively in the related literature; the full Drebin dataset [3] dated from 2010 to 2012, a collection (subset) from VirusShare [32] dated from 2014 to 2017, as well as a collection from Androzoo [33] dated from 2017 to 2020. Precisely, the number of malware samples taken from each of the aforementioned datasets was correspondingly 5560, 10,000, and 1520, that is, a total of 17,080 samples. It is important to note that the Drebin dataset contains outdated samples, and therefore, these pieces of malware were only used to cross-validate our results vis-à-vis the related work. On the other hand, Androzoo samples are expected to be more challenging when it comes to detection, given that this corpus contains newer, and therefore more sophisticated malware in comparison to those in the Drebin and VirusShare datasets.

After analysing 17,080 mobile malware samples from the three datasets, a total of more than 50,000 different vectors of features were collected. Note that each vector is a binary representation of each distinct feature. Let us for example take the simplest case of just two apps. App A1 uses permission A, intent B, and initiates communication via HTTP with IP: 1.1.1.1. App A2 on the other hand, uses permission B, intent C, and triggers communication via HTTPS with IP: 1.1.1.2. As a result, as shown in Table 2, the vectors of features for both app A1 and app A2 have twice as many binary indicators. Of course, a real-world app would produce a much more lengthy vector.

Table 2. Feature vectors for example apps A1 and A2.

App	permissionA	permissionB	intentA	intentB	HTTP	HTTPS	1.1.1.1	1.1.1.2
A1	1	0	1	0	1	0	1	0
A2	0	1	0	1	0	1	0	1

Naturally, putting aside the capacity of the data mining tool, the overall number of 50,000 different vectors, meaning 50,000 diverse columns (features), exceeds the ability of Androtomist and virtually and proof-of-concept tool to fetch them in a reasonable time using SQL queries. So, for the needs of our experiments in Section 4.2, a smaller subset of 100 or 120 malware samples per dataset was finally used. Specifically, we randomly selected 100 malware samples from each of the Drebin and VirusShare datasets and 120 from—the more challenging—AndroZoo. This sample reduction technique is also observed in previous work, including those in References [11,15,18]. Specifically, common malware families can produce overestimated results. Therefore, we filtered commonplace information among malware families, such as sample year and package name from each dataset, for the sake of obtaining equally distributed samples across the different malware families. It is important to mention that all the three (reduced) datasets are balanced, containing an equal number of randomly selected malicious and benign apps. The benign apps were collected from Google Play Store [34] and are common across all the three datasets but the AndroZoo, which incorporates 20 more.

3.4. Classifiers and Metrics

The vast majority of the state-of-the-art methods focused on the detection of malicious samples were ranked based on recall and precision of correct answers combined by the F1 measure. For each instance separately, these measures of correctness are concentrated on providing a binary answer, either a positive (malware class) or a negative (goodware class) one. This indicates that the information about the distribution of positive and negative instances is necessary in order to set this threshold. In this paper, we follow exactly the same evaluation procedure to achieve compatibility of comparison with previously reported results. For each dataset, the set of the extracted scores based on the test instances are normalized in the interval of [0,1] per classification model per examined method. To this direction, the estimation of the threshold is set equal to 0.5.

Moreover, we use the AUC of the receiver-operating characteristic curve as the main evaluation measure [35]. Generally, AUC quantifies the effectiveness of each examined approach for all possible score thresholds. As a rule, the value of AUC is extracted by examining the ranking of scores rather than their exact values produced when a method is applied to a dataset. Noticeably, the estimation of the AUC measure is based on all possible thresholds. Besides, this evaluation measure does not depend on the distribution of positive and negative samples.

In this paper, a sizable number of well-known and popular supervised ML algorithms are applied. That is, for each dataset, we consider seven of the most widely used binary classification models, namely Logistic Regression (LR), Naïve Bayes, Random Forest, AdaBoost, Support-vector machine (SVM), k-nearest neighbors (k-NN) and Stochastic Gradient Descent (SGD). The majority of the classification algorithms applied falls under eager learning. In this category, supervised learning algorithms attempt to build a general model of the malicious instances, based on the training data. Obviously, the performance of such classifiers strongly depends on the size, quality and representative of the training data. On the other hand, k-NN is a weak learner (known as lazy learner) as that is not using the training data to construct a general model, but it makes a decision based on information extracted for each sample separately. For each classifier applied, the default values of the parameter settings are used. The general model of each eager classifier is built following the 10-fold cross-validation technique. In this technique, the original dataset is randomly partitioned into 10 equal sized sub-datasets. A single sub-dataset is retained for the testing, while the remaining 9 are used for training. This process is repeated 10 times, and each time using a different sub-dataset for testing. Taking into consideration that a set of 100 samples are handled in each of the three examined datasets, a percent of 90% is used as training and 10% as test set per time. The results are then averaged to produce a single estimation.

Finally, for each dataset, a simple meta-model, that is, an aggregate function, is developed combining all the base classifiers applied either in hybrid or static methods separately. This heterogeneous ensemble is based on the average score resulted by all seven binary classification models for each sample. Regarding the Android malware literature, “ensemble learning” is also exploited by the authors of References [36,37].

4. Evaluation

We assess the detection capabilities of Androtomist using both signature and anomaly-based detection. In fact, regarding signature-based detection, Androtomist has the capacity to automatically generate detection signatures by combining static and dynamic features. We first present the signature-based results in Section 4.1, and then we detail on the classification results after training a ML model with the identical set of features.

4.1. Signature-Based Detection

Generally, misuse detection, also known as signature-based detection, relies on known signatures, that is, detection rules aiming to discern between benign and malicious pieces of code. While these systems are capable of identifying previously encountered malicious software and may have a high degree of portability between platforms, they miss to recognize novel instances of malware or variations of known ones. Thus, the detection ability of a misuse detection system, as the one examined in this subsection, primarily depends on the newness of the detection rules the system has been configured with.

In the context of Androtomist, this type of detection involves comparing the features collected during static and dynamic analysis for a new (“unknown”) app against those already stored in the DB for all the malicious apps encountered so far. Put differently, to assess signature-based detection against the three datasets mentioned in Section 3.3, a sufficient mass of malware samples had to be analyzed beforehand to create signatures.

Therefore, by using the sample reduction technique explained in Section 3.3, 1902 random malware samples were chosen (correspondingly 499, 1403 from Drebin, and VirusShare) and being both statically and dynamically analyzed to collect features. This number pertains to those malware

samples that (a) are not included in the reduced datasets presented in Section 3.3, and (b) have been filtered to prune duplicate malware families by using each app’s package name and hash signature.

The derived signatures comprise combinations of features collected from both static and dynamic analysis. A combination incorporates at least one feature from minimum two different categories of features, namely permissions, intents, API calls, network traffic, java classes, and triggered intents. When a new app is analyzed, Androtomist checks for a possible signature match. If the analyzed app yields results that match with two or more signatures, then the app is flagged as malicious.

On top of that, individual features which are spotted regularly among malware, but rarely among goodware are also being labeled as suspicious in the Androtomist’s analysis report. Specifically, as an indication, the left column of Table 3 lists the top 15 most frequent permissions detected in the utilized set of 1902 malware samples. The right column, on the other hand, contains another 15 permissions which were observed in the same set of samples, but they were scarce or nonexistent in the set of 120 goodware apps selected from Google Play Store. Both these columns are ordered by percentage of occurrence in the set of 1902 pieces of malware. Finally, Table 4 contains the 15 most frequent permissions detected in the utilized sample of 120 apps downloaded from Google Play.

Table 3. Top 30 suspicious permissions.

Most Common in Malware	%	Least Common in Goodware	%
android.permission.INTERNET	96.2	com.android.browser.permission.READ_HISTORY_BOOKMARKS	12.2
android.permission.READ_PHONE_STATE	91.8	com.android.browser.permission.WRITE_HISTORY_BOOKMARKS	11.9
android.permission.WRITE_EXTERNAL_STORAGE	85.7	android.permission.WRITE	11.4
android.permission.ACCESS_NETWORK_STATE	59.0	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS	9.8
android.permission.ACCESS_WIFI_STATE	46.3	android.permission.INSTALL_PACKAGES	5.8
android.permission.ACCESS_COARSE_LOCATION	42.6	com.android.launcher.permission.READ_SETTINGS	5.5
android.permission.ACCESS_FINE_LOCATION	37.7	android.permission.MOUNT_UNMOUNT_FILESYSTEMS	5.4
android.permission.WAKE_LOCK	26.5	android.permission.RESTART_PACKAGES	4.5
android.permission.VIBRATE	25.3	android.permission.WRITE_APN_SETTINGS	4.1
android.permission.RECEIVE_BOOT_COMPLETED	24.2	android.permission.CHANGE_CONFIGURATION	2.2
com.android.launcher.permission.INSTALL_SHORTCUT	23.3	android.permission.WRITE_SECURE_SETTINGS	1.8
android.permission.CHANGE_WIFI_STATE	19.9	android.permission.ACCESS_COARSE_UPDATES	1.7
android.permission.CALL_PHONE	16.9	android.permission.DELETE_PACKAGES	1.5
android.permission.GET_TASKS	14.9	android.permission.READ_SETTINGS	0.8
android.permission.SEND_SMS	14.8	android.permission.RECEIVE_WAP_PUSH	0.5

Table 4. Top 15 permissions in goodware.

Most Common in Goodware	%
android.permission.INTERNET	100%
android.permission.ACCESS_NETWORK_STATE	96%
android.permission.RECEIVE_BOOT_COMPLETED	86%
android.permission.GET_ACCOUNTS	85%
android.permission.USE_CREDENTIALS	85%
android.permission.WRITE_EXTERNAL_STORAGE	85%
android.permission.WAKE_LOCK	82%
com.google.android.c2dm.permission.RECEIVE	78%
android.permission.MANAGE_ACCOUNTS	77%
android.permission.CAMERA	76%
android.permission.AUTHENTICATE_ACCOUNTS	76%
android.permission.READ_EXTERNAL_STORAGE	72%
android.permission.WRITE_SYNC_SETTINGS	70%
android.permission.READ_SYNC_SETTINGS	70%
android.permission.ACCESS_WIFI_STATE	68%

Furthermore, dynamic instrumentation reveals a certain pattern in Java classes used among malware. Especially, several malware apps, that is, correspondingly 9%, 2%, and 1% of the instances in

our AndrooZoo, VirusShare, and Drebin datasets, instantiate the *DexClassLoader* class [38]. This class is used to load other classes from *jar* and *apk* type of files containing a *classes.dex* entry. This class loader can be exploited to execute code which is not installed as part of the app per se. Another interesting observation is that the *java.net.Socket* [39] class was found quite frequently in malware instances, that is, 15% in AndrooZoo, and 10% in VirusShare and Drebin.

For the sake of comparison, Table 5 summarizes the results obtained when the detection engine is (a) fed only with signatures created during static analysis, and (b) signatures created during hybrid analysis. As observed, when the features of static analysis are fused with those acquired via dynamic instrumentation, the detection rate is correspondingly increased by 7.5%, 3%, 6% for “unknown” apps in the AndroZoo, VirusShare, and Drebin dataset. Obviously, this is because the number and variety of features have increased, thus allowing more of them to match against malicious signatures.

Table 5. Signature-based detection scores (%).

Dataset	True Positive Rate (TPR)		Improvement Rate (Hybrid)
	Static	Hybrid	
AndroZoo	86.6	94.1	+7.5
VirusShare	96	99	+3
Drebin	88	94	+6

4.2. Anomaly-Based Detection

The vectors of features derived from both static and dynamic analysis over the datasets described in Section 3.3 were utilized to conduct anomaly-based detection. To do so, we exploited the open source data mining tool Orange [40]. As detailed in Section 3.4, we employed seven ML algorithms used extensively in the related literature. The selected sampling technique was 10-fold cross-validation. We consider the following classification performance metrics, where TP, TN, FP, and FN represent correspondingly true positives, true negatives, false positives, and false negatives.

- Accuracy (CA) : $\frac{TP+TN}{TP+TN+FP+FN}$. The number of correctly classified patterns over the total number of patterns in the sample.
- Precision (P) : $\frac{TP}{TP+FP}$. The ratio of TP values over the sum of TP and FP.
- Recall (R) : $\frac{TP}{TP+FN}$. The ratio of TP over the sum of TP and FN.
- Area Under Curve (AUC): The higher positive-over-negative value ranking capability of a classifier.
- $F1 : 2 \frac{P \cdot R}{P+R}$.

Table 6 provides a comparative overview of the classification performance scores yielded from static analysis against those obtained from hybrid analysis. The best score per metric in the table is underlined. It seems that the proposed approach based on hybrid analysis constantly outperforms the static method when the performance of AUC as well as the effectiveness of binary measures (CA, P, R, and F1) is considered. As can be observed, in almost all cases of all three datasets, hybrid analysis is more effective than static one; only in Drebin corpora static outperforms hybrid when Naïve Bayes and LR models are applied. This shows that the proposed hybrid analysis is clearly a better option than static one.

Logistic Regression and AdaBoost hybrid models seem to be the overall best performing models for AndroZoo dataset. Moreover, a larger number of the examined classifiers, namely LR, Random Forest, AdaBoost and SVM help hybrid to achieve exceptional results in comparison to static method for the VirusShare dataset. However, the vast majority of the examined hybrid models seems to be exceptionally successful (the only exception is the Naïve Bayes model) in the Drebin dataset. In general, the hybrid approach achieves more stable performance across all three datasets in comparison to the method based on static analysis. It is also remarkable that the biggest improvement of hybrid models is achieved in AndroZoo corpus. It is noticeable that the improvement in performance in

terms of AUC of the best hybrid models with respect to that of the best static ones is higher than 10% when the AndroZoo corpora is considered. All these indicate that the hybrid approach is much more reliable and effective in more challenging malware cases where there are new and more sophisticated malware conditions.

Table 6. Results per dataset and classification performance metric.

Dataset	ML Classifier	AUC		CA		F1		Precision		Recall	
		Static	Hybrid	Static	Hybrid	Static	Hybrid	Static	Hybrid	Static	Hybrid
AndroZoo	Logistic Regression	0.870	<u>0.978</u>	0.888	0.888	0.894	0.888	0.906	0.890	0.888	0.888
	Naïve Bayes	0.777	0.941	0.811	0.854	0.825	0.854	0.852	0.855	0.811	0.854
	Random Forest	0.938	0.972	0.895	0.888	0.900	0.888	0.910	0.889	0.895	0.888
	k-NN	0.914	0.954	0.867	0.871	0.868	0.871	0.869	0.874	0.867	0.871
	AdaBoost	0.915	0.971	0.909	0.901	0.913	0.901	<u>0.923</u>	0.901	0.909	0.901
	SGD	0.808	0.918	0.881	<u>0.918</u>	0.882	<u>0.918</u>	0.883	0.918	0.881	<u>0.918</u>
	SVM	0.900	0.930	0.846	0.850	0.856	0.849	0.880	0.858	0.846	0.850
VirusShare	Logistic Regression	0.993	<u>1.000</u>	0.975	<u>1.000</u>	0.975	<u>1.000</u>	0.975	<u>1.000</u>	0.975	<u>1.000</u>
	Naïve Bayes	0.975	0.997	0.910	0.965	0.910	0.965	0.910	0.967	0.910	0.965
	Random Forest	0.994	<u>1.000</u>	0.975	0.995	0.975	0.995	0.975	0.995	0.975	0.995
	k-NN	0.990	0.995	0.980	0.990	0.980	0.990	0.981	0.990	0.980	0.990
	AdaBoost	0.965	<u>1.000</u>	0.965	<u>1.000</u>	0.965	<u>1.000</u>	0.965	<u>1.000</u>	0.965	<u>1.000</u>
	SGD	0.955	0.990	0.955	0.990	0.955	0.990	0.957	0.990	0.955	0.990
	SVM	0.983	<u>1.000</u>	0.950	0.995	0.950	0.995	0.955	0.995	0.950	0.995
Drebin	Logistic Regression	0.998	0.990	0.989	0.989	0.989	0.989	0.989	0.989	0.989	0.989
	Naïve Bayes	0.986	0.970	0.923	0.967	0.924	0.967	0.924	0.967	0.923	0.967
	Random Forest	0.998	<u>1.000</u>	0.984	0.995	0.984	0.995	0.984	0.995	0.984	0.995
	k-NN	0.982	0.995	0.962	0.995	0.962	0.995	0.965	0.955	0.962	0.995
	AdaBoost	0.973	<u>1.000</u>	0.973	<u>1.000</u>	0.973	<u>1.000</u>	0.973	<u>1.000</u>	0.973	<u>1.000</u>
	SGD	0.990	<u>1.000</u>	0.989	<u>1.000</u>	0.989	<u>1.000</u>	0.989	<u>1.000</u>	0.989	<u>1.000</u>
	SVM	0.999	<u>1.000</u>	0.984	<u>1.000</u>	0.984	<u>1.000</u>	0.984	<u>1.000</u>	0.984	<u>1.000</u>

The version of our method based on static analysis is also very effective achieving the best results in VirusShare as well as Drebin corpus. As concerns the AndroZoo dataset, static analysis performs poorly. Figures A1a–c and A2a–c in the Appendix A depict the effectiveness of ROC curves when the LR classifier is applied on each dataset for the presented version of the hybrid as well as the version of static analysis, respectively.

5. Discussion

Regarding signature-based detection, the scores in Table 5 suggest that when static analysis is combined with dynamic instrumentation into a hybrid approach, the detection rate can be augmented by at least 3% and up to 7.5%. Another salient observation is that the classification performance differs significantly among the three datasets. Specifically, as observed from the same Table, the detection rate vary considerably from 87% to 96% for static analysis, but this divergence is much lesser, from 94% to 99%, for the hybrid scheme. This also support the view that hybrid analysis by means of dynamic instrumentation leads to more robust classification results. Particularly, the detection rate for—the more challenging—AndroZoo dataset, is the lowest among all scores when using only static analysis, but it augments by +7.5% when is backed up by the results of dynamic instrumentation. The same observation stands true for the VirusShare or Drebin datasets. This results actually do not come as a surprise, and is verified by relevant works in the literature [41].

The same general inference is deduced from Table 6 with respect to the results obtained from anomaly-based detection. That is, in their vast majority, the scores regarding the most two important classification performance metrics, namely AUC and F1, are superior when hybrid analysis is involved, irrespective of the dataset and the ML classifier used.

To further demonstrate the usefulness of the examined approaches, Figure 4 depicts the evaluation results (AUC) of hybrid and static analysis in the employed datasets when applying the seven classifiers. As can be clearly seen, the basic patterns are the same across all the datasets. The proposed hybrid method is more effective, surpassing the approach based on static analysis in almost all cases. As already pointed out in Section 4.2, the only exceptions are LR and Naïve bays in the Drebin dataset. In both VirusShare and Drebin corpora, the static method is also very effective in all cases.

With respect to the results on AndroZoo corpora, the results suggest that the use of hybrid analysis helps the approach to become more stable. As can be seen, the proposed hybrid approach outperforms in all cases the static method, by a large margin (more than 10%). Recall from Section 3.3 that the AndroZoo dataset embraces newer instances of malware in contrast to VirusShare and Drebin ones. It seems therefore that this challenging condition significantly affects the performance of static models. The same patterns are provided in case where F1 measure is considered as it is illustrated in Figure A3 in the Appendix A.

In addition, we also consider the combination of the base classification models (seven classifiers) by averaging their answers per sample on each dataset separately. Figure 5 demonstrates the percentage in performance (AUC) of both these ensemble methods based on the hybrid and static analysis, respectively. Especially, vis-à-vis the results of Table 6, the effectiveness of both the ensemble approaches in terms of AUC, is better than that of any single base classifier in the vast majority of the cases. Although, the version of static analysis is effective enough, the version based on hybrid analysis consistently outperforms its performance. Overall, as can be observed, the proposed hybrid ensemble is the most effective one surpassing that of static analysis and improving the reported results of all base classifiers across all the three datasets considered. With respect to the results on AndroZoo corpora, the proposed hybrid ensemble clearly seems to be the top-performing model which achieves the best results not only among all base models, but also in comparison with the static one. The improvement in average performance of the hybrid ensemble model with respect to that of the static is higher than 4%.

Table 7 demonstrates the improvement in performance, that is, difference of AUC scores, of the individual and ensemble models when the hybrid analysis is considered in comparison to the case where only the static one is used. Statistical significance of these differences is estimated using an approximate randomization test [42]. The null hypothesis is that there is no difference between the two cases and we reject this hypothesis when $p < 0.05$. As can be seen, in most of the cases, the models extracted from the hybrid analysis are significantly ($p < 0.05$) better than static ones. Notable exception is Naïve Bayes on Drebin dataset where the variance with the corresponding static model is slightly decreased. In AndroZoo dataset, all hybrid base models are more improved than the corresponding static ones as well as the hybrid ensemble model gains more than the static one. This indicates that hybrid analysis is superior, able to handle challenging datasets and better suits more sophisticated malware.

On the other hand, the differences between the hybrid and static ensembles are not statistically significant on the VirusShare and Drebin data sets. As concerns the individual models, in the first corpus, the results of Naïve Bayes, AdaBoost and SGD are improved the most, while in the latter only the difference in results of k-NN and SGD models appears to be significant. The betterment of base models based on static analysis seems not to be correlated with the relative increase in the number of malware samples used to extract the static classification models. Apparently, an increasing number of malware instances on AndroZoo corpus does not help Static analysis to achieve better performance.

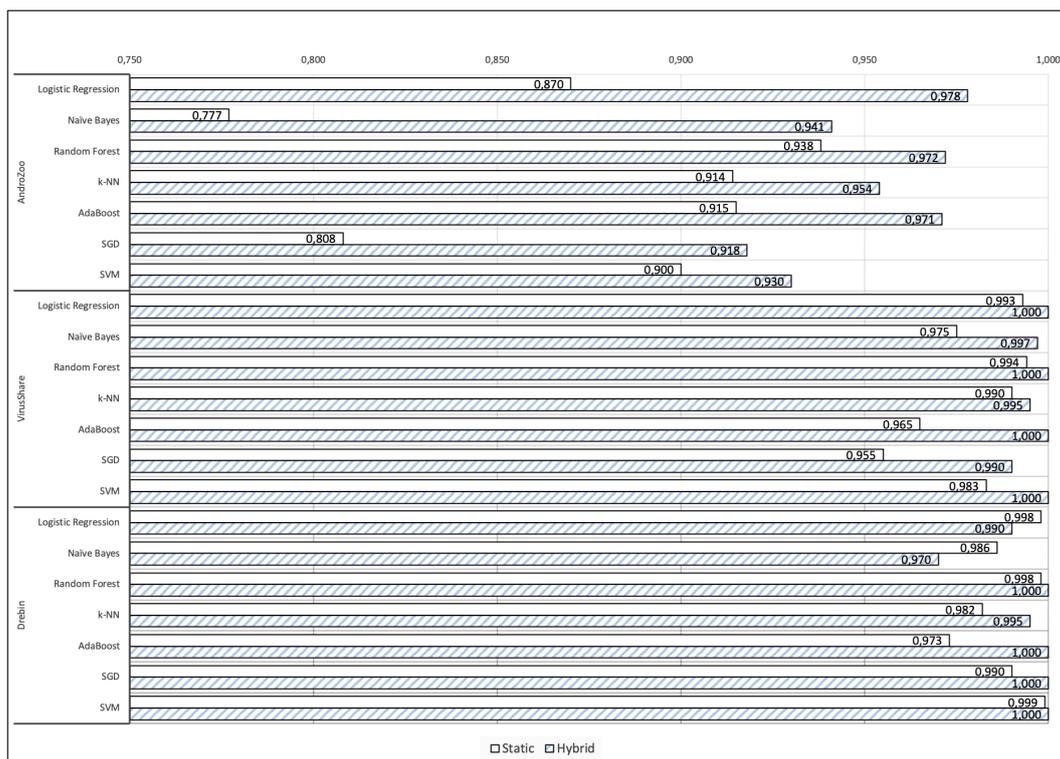


Figure 4. The performance (AUC) of the proposed static and hybrid analysis on AndroZoo, VirusShare, and Drebin corpora for different classification models.

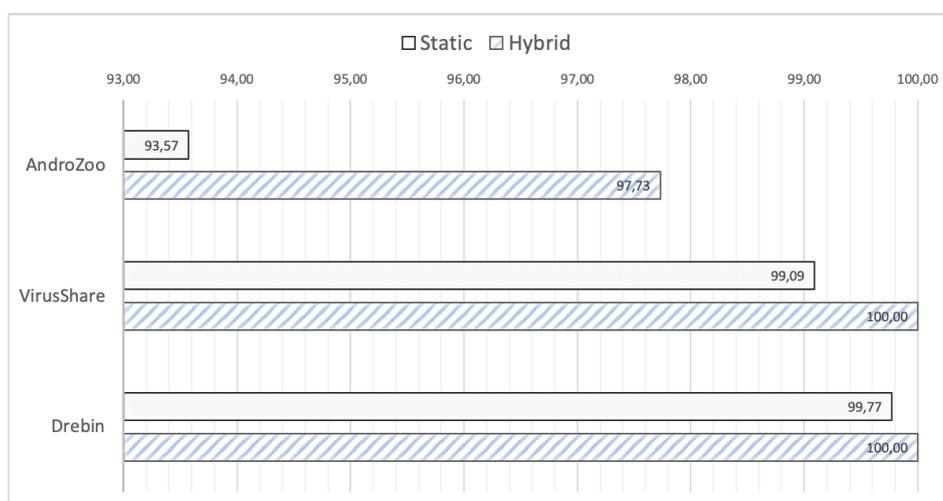


Figure 5. The percentage in performance (AUC) of hybrid and static methods by averaging the output of the base classifiers on the three datasets

Table 7. Improvement in performance (difference in AUC) between methods using hybrid analysis (base models and ensemble) and static analysis (base models and ensemble). Statistically significant differences ($p < 0.05$) are indicated in boldface.

Data Set	LR	Naïve Bayes	Random Forest	k-NN	AdaBoost	SGD	SVM	Ensemble
AndroZoo	0.108	0.164	0.034	0.040	0.056	0.110	0.030	0.042
VirusShare	0.007	0.022	0.006	0.005	0.035	0.035	0.017	0.009
Drebin	-0.008	-0.016	0.002	0.013	0.027	0.010	0.001	0.002

A key factor that affects the performance of malware detection methods is the importance of features contained in malware samples. To study which feature matters more for detecting malware in

both the static and hybrid method, we concentrate on the Logistic Regression model, which according to Table 6, seems to be the top performer across all datasets. To this end, Figure 6 illustrates the average feature importance scores on all three datasets when Logistic Regression is applied for both hybrid and static method. As feature importance scores have been directly used the average coefficients of each feature category explored following a linear regression algorithm. More specifically, the features importance scores are assigned by coefficients calculated as part of a linear regression model per set of features.

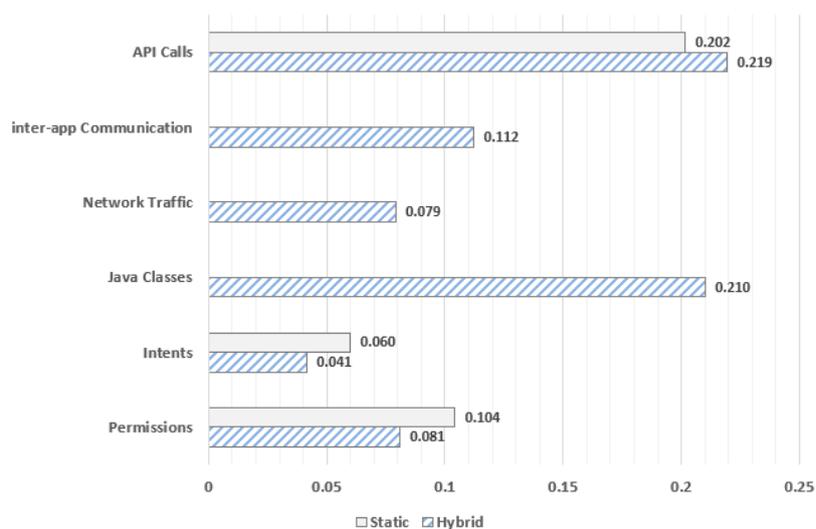


Figure 6. Average Feature Importance scores of static and hybrid analysis on all three datasets for a varying set of feature categories.

As already mentioned, six categories of features are explored in total; permissions, intents, API calls, network traffic, inter-app communication and Java classes, where only the first three apply to static analysis. In general, from Figure 6 it is observed that the API calls category is more influential and tends to improve both static and hybrid methods. It is also noteworthy that the category of Java classes achieves also significantly high average importance scores enhancing the effectiveness of hybrid method on all three datasets. On the other hand, the contribution of both intents and permissions categories is lower not only in static, but also in hybrid method. Lastly, network traffic is a category that includes features with slightly better coefficients than intents and permissions categories when hybrid analysis is considered. The case of inter-app communication category seems also to be of particular importance for hybrid method. However, its average coefficient scores similar to intents and permissions categories are significantly smaller than those of both API calls and Java classes categories on hybrid approach.

For the sake of completeness, Table 8 provides a comparison of the best-case accuracy performance between Bridemaid [13], HADM [14], SAMADroid [16], Surendran et al. [20], and Androtomist. The comparison reveals that in all the applicable cases Androtomist is able to improve the best results per matching dataset. Precisely, Androtomist surpasses the rest of the state-of-the-art detection systems when using malware from the Drebin or VirusShare datasets. A direct comparison with Bridemaid is not feasible because this system was evaluated using a mixture of malware samples from different datasets, that is, *Genome*, which is not shared anymore, and *Contagio* which is considered obsolete. Naturally, as a rule of thumb, the performance of any detection model depends on the malware samples used during both its evaluation and testing phases [43].

For easy reference, the same Table lists the different kind of features collected during the analysis phase per detection system. Recall from Section 3.1 that during static analysis, Androtomist gathers the most common features seen in the relevant work, namely permissions, intents, and API calls. The same static features are collected in [14,16]. On the other hand, for dynamic analysis, Androtomist

relies on dynamic instrumentation, enabling the extraction of a wide variety of features. Contrariwise to other schemes included in Table 8, system calls are consciously neglected as a feature, since its exploitation has been significantly restricted from Android v.7.0 onward [44]. Precisely, similar to SELinux [45], Android is anymore based on *Seccomp* kernel module [46] to further limit access to the kernel by blocking certain system calls. Put simply, starting from Android v.7.0, *Seccomp* kernel has been applied to processes in the media frameworks. Moreover, as of v.8.0, a *Seccomp* filter is installed into the *zygote*, that is, the forking handling process from which all Android apps are derived. *Seccomp* blocks access to certain system calls, such as *swapon/swapoff*, which were the root cause for certain security attacks [47]. Additionally, *Seccomp* blocks key control system calls, which are not useful to apps. Therefore, when analysing newer Android apps, as those in the *AndroZoo* dataset, the merit of system calls as a feature in malware detection schemes has been greatly diminished. Last but not least, some of these tools such as *SAMADroid* are intended to run on the smartphone. In this case, the app's features are locally collected when the user interacts with it. Then, the collected features are sent to a remote host in which a decision is made about the app. *Androtomist* on the other hand only runs on a remote host, and therefore the app must be analysed prior to its installation.

Table 8. Comparison of state-of-the-art hybrid systems in terms of collected features and classification accuracy (best case). “Mixed” means a mixed, but not strictly defined dataset, containing records from *Drebin*, *Genome*, and *Contagio* datasets.

Detection System	Groups of Features Collected	Accuracy (AC) Achieved Per Dataset			
		Drebin	Virusshare	AndroZoo	Mixed
Bridemaide [13]	n-grams classification, SMS, System calls, Admin privileges	N/A	N/A	N/A	99.70%
HADM [14]	API calls, Permissions, Intents, System calls	N/A	94.70%	N/A	N/A
SAMADroid [16]	API calls, Permissions, Intents, System calls	99.07%	N/A	N/A	N/A
Surendran et al. [20]	Permissions, API calls, System calls	99.00%	N/A	N/A	97.00%
Androtomist	API calls, Permissions, Intents, Network traffic, Java classes, Inter-process communication	100%	100%	91.80%	N/A

6. Conclusions

This paper presents *Androtomist*, an automated and configurable hybrid analysis tool, which combines static analysis with dynamic instrumentation to analyse app behavior in the Android platform. The results over three different datasets show that this dual analysis of mobile apps can significantly enhance the detection capabilities of a detection model. Furthermore, a comparison between static and hybrid analysis by means of instrumentation reveals asymmetry, that is, the latter is able to yield better classification results even throughout diverse datasets. By collating the performance of our proposal vis-à-vis similar state-of-the-art detection systems we demonstrate its superiority. Ultimately, *Androtomist* not only offers an easy-to-use environment for virtually everyone to analyze mobile apps, but it is also configurable to the point where researchers can import custom dynamic instrumentation hooks and scripts.

In the current work, both the proposed approaches use a randomly chosen subset of malware samples per dataset, a typical choice in similar studies. It could be interesting to investigate how this selection process can be optimized by means of clustering toward including a set of the best possible representative malware samples showing a common behavioural pattern. Both the heterogeneous ensemble approaches rely on base models with default parameter settings. This could be used to further enrich the pool of our base verifiers considering several versions of the same approach with different fixed and tuned parameter settings. Another future work direction could focus on combining the methods based on hybrid and static analysis in a more complex approach.

Author Contributions: Conceptualization, V.K. and G.K.; methodology, V.K., G.K., D.G., and N.P.; validation, V.K., G.K., D.G., and N.P.; investigation, V.K.; software, V.K.; writing-original draft preparation, V.K., G.K., and N.P.; writing-review and editing, V.K., G.K., N.P., and D.G.; supervision, G.K., D.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AUC	Area under the curve
APP	Application
ML	Machine Learning
API	Application programming interface
TAN	Tree Augmented Naive Bayes
VM	Virtual Machine
DB	Database
ADB	Android Debug Bridge
APK	Android application package
OS	Operating System
CSV	Comma-separated values
SVM	Support-vector machine
SGD	Stochastic Gradient Descent
ROC	Receiver operating characteristic curve

Appendix A

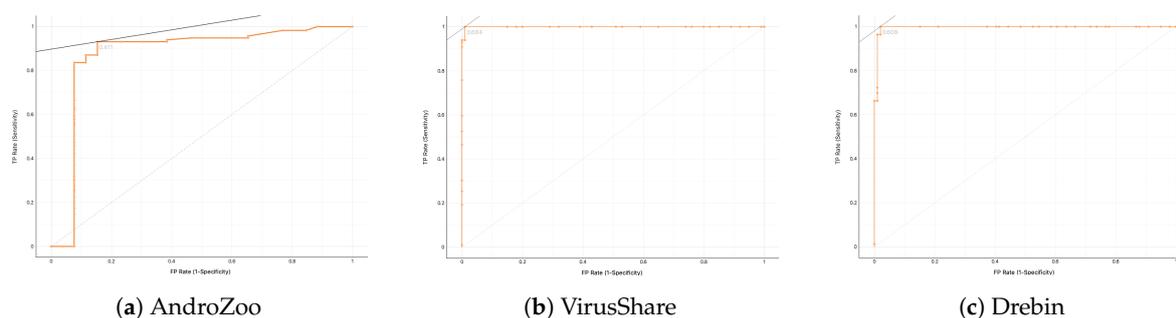


Figure A1. ROC curves (Static analysis).

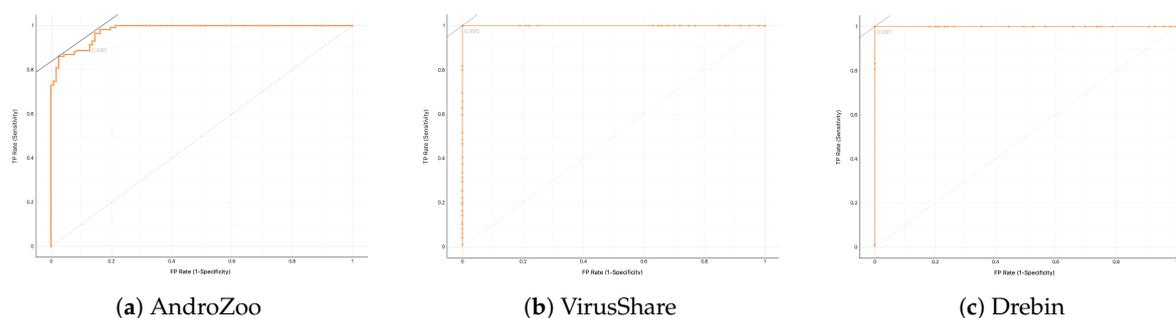


Figure A2. ROC curves (Hybrid analysis).

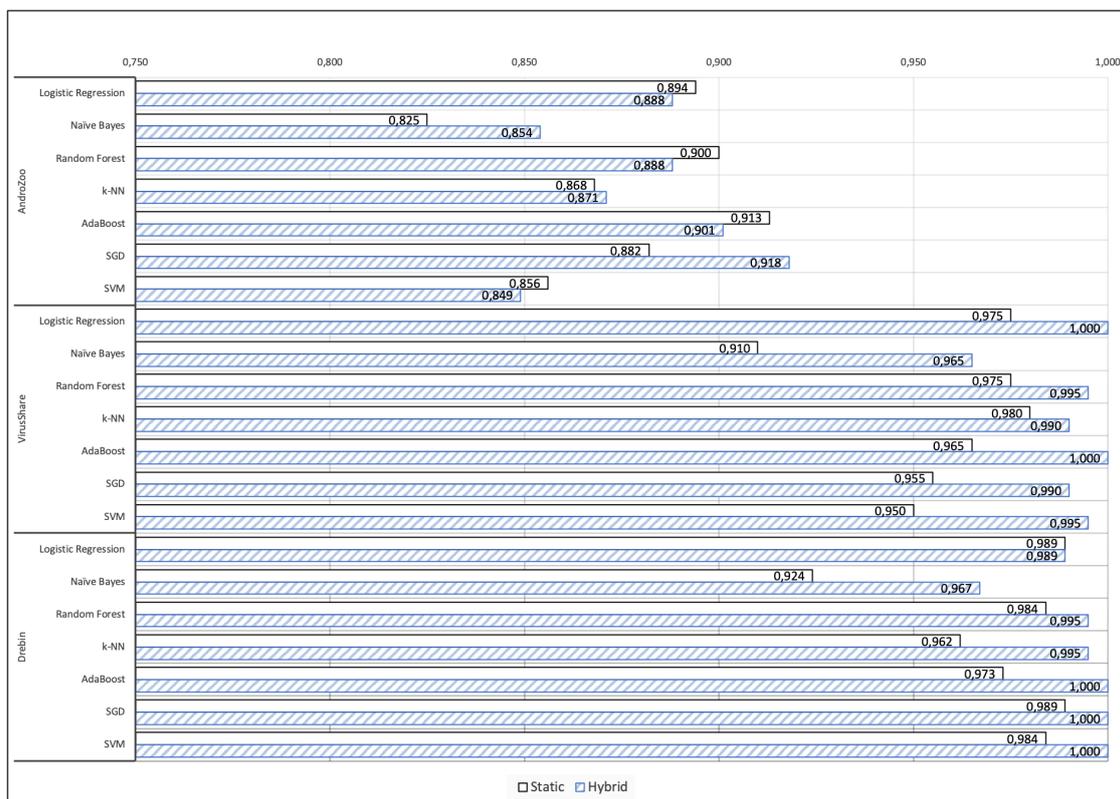


Figure A3. The performance (F1) of the proposed static and hybrid analysis on AndroZoo, VirusShare, and Drebin corpora for different classification models.

References

1. Mobile Threat Report. Available online: <https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf> (accessed on 20 May 2020)
2. Aafer, Y.; Du, W.; Yin, H. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *International Conference on Security and Privacy in Communication Systems*; Springer: Cham, Switzerland, 2013.
3. Arp, D.; Spreitzenbarth, M.; Gascon, H.; Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*; The Internet Society: Reston, VA, USA, 2014.
4. Wang, W.; Wang, X.; Feng, D.; Liu, J.; Han, Z.; Zhang, X. Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection. *IEEE Trans. Inf. Forensics Secur.* **2014**, *9*, 1869–1882. [[CrossRef](#)]
5. Damopoulos, D.; Kambourakis, G.; Portokalidis, G. The Best of Both Worlds: A Framework for the Synergistic Operation of Host and Cloud Anomaly-Based IDS for Smartphones. In *Proceedings of the Seventh European Workshop on System Security (EuroSec)*, Amsterdam, The Netherlands, 13 April 2014.
6. Damopoulos, D.; Kambourakis, G.; Gritzalis, S.; Park, S. Exposing mobile malware from the inside (or what is your mobile app really doing?). *Peer-to-Peer Netw. Appl.* **2014**, *7*, 687–697. [[CrossRef](#)]
7. Papamartzivanos, D.; Damopoulos, D.; Kambourakis, G. A cloud-based architecture to crowdsource mobile app privacy leaks. In *proceedings of the 18th Panhellenic Conference on Informatics*, Athens, Greece, 2–4 October 2014; pp. 1–6.
8. Kouliaridis, V.; Barmatsalou, K.; Kambourakis, G.; Chen, S. A Survey on Mobile Malware Detection Techniques. *IEICE Trans. Inf. Syst.* **2020**, *103*, 204–211. [[CrossRef](#)]
9. Odusami, M.; Abayomi-Alli, O.; Misra, S.; Shobayo, O.; Damasevicius, R.; Maskeliunas, R. Android Malware Detection: A Survey. In *International Conference on Applied Informatics*; Springer: Cham, Switzerland, 2018; pp. 255–266.

10. Patel, K.; Buddadev, B. Detection and Mitigation of Android Malware Through Hybrid Approach. In *International Symposium on Security in Computing and Communication*; Springer: Cham, Switzerland, 2015; pp. 455–463.
11. Cam, N.T.; Pham, V.H.; Nguyen, T. Detecting sensitive data leakage via inter-applications on Android using a hybrid analysis technique. *Clust. Comput.* **2017**, *22*, 1055–1064. [[CrossRef](#)]
12. Tuan, L.H.; Cam, N.T.; Pham, V.H. Enhancing the accuracy of static analysis for detecting sensitive data leakage in Android by using dynamic analysis. *Clust. Comput.* **2017**, *22*, 1079–1085. [[CrossRef](#)]
13. Martinelli, F.; Mercaldo, F.; Saracino, A. Bridemaid: An Hybrid Tool for Accurate Detection of Android Malware. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS 17), Abu Dhabi, UAE, 2–6 April 2017.
14. Xu, L.; Zhang, D.; Jayasena, N.; Cavazos, J. HADM: Hybrid Analysis for Detection of Malware. In *Proceedings of the SAI Intelligent Systems Conference; 2016 Lecture Notes in Networks and Systems*; Springer: Cham, Switzerland, 2018; pp. 702–724.
15. Ali-Gombe, I.; Saltaformaggio, B.; Ramanujam, J.R.; Xu, D.; Richard, G.G. Toward a more dependable hybrid analysis of android malware using aspect-oriented programming. *Comput. Secur.* **2018**, *73*, 235–248. [[CrossRef](#)]
16. Arshad, S.; Shah, M.A.; Wahid, A.; Mehmood A.; Song H.; Yu, H. SAMADroid: A Novel 3-Level Hybrid Malware Detection Model for Android Operating System. *IEEE Access* **2018**, *6*, 4321–4339. [[CrossRef](#)]
17. Tsutano, Y.; Bachala, S.; Srisa-An, W.; Rothermel, G.; Dinh, J. Jitana: A modern hybrid program analysis framework for android platforms. *J. Comput. Lang.* **2019**, *52*, 55–71. [[CrossRef](#)]
18. Wang, X.; Yang, Y.; Zhu, S. Automated Hybrid Analysis of Android Malware Through Augmenting Fuzzing With Forced Execution. *IEEE Trans. Mob. Comput.* **2019**, *12*, 2768–2782. [[CrossRef](#)]
19. Fang, Q.; Yang, X.; Ji, C. A Hybrid Detection Method for Android Malware. In Proceedings of the 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference, Chengdu, China, 15–17 March 2019.
20. Surendran, R.; Thomas, T.; Emmanuel, S. A TAN based hybrid model for android malware detection. *J. Inf. Secur. Appl.* **2020**, *54*, 102483. [[CrossRef](#)]
21. Amaaas. Available online: <https://amaaas.com/> (accessed on 20 May 2020).
22. VirusTotal. Available online: <https://www.virustotal.com/> (accessed on 20 May 2020).
23. Androtomist. Available online: <https://androtomist.com/> (accessed on 20 May 2020).
24. VirtualBox. Available online: <https://www.virtualbox.org/> (accessed on 20 May 2020).
25. Android Debug Bridge. Available online: <https://developer.android.com/studio/command-line/adb> (accessed on 20 May 2020).
26. Kouliaridis, V.; Barmapsalou, K.; Kambourakis, G.; Wang, G. Mal-Warehouse: A Data Collection-as-a-Service of Mobile Malware Behavioral Patterns. In Proceedings of the 2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, Guangzhou, China, 8–12 October 2018.
27. Androtomist's Source Code. Available online: <https://github.com/billkoul/Androtomist> (accessed on 20 May 2020).
28. European Union Public Licence. Available online: https://ec.europa.eu/info/european-union-public-licence_en (accessed on 20 May 2020).
29. Apktool. Available online: <https://ibotpeaches.github.io/Apktool/> (accessed on 20 May 2020).
30. Frida. Available online: <https://frida.re/> (accessed on 20 May 2020).
31. Monkey. Available online: <https://developer.android.com/studio/test/monkey> (accessed on 20 May 2020).
32. Virus Share. Available online: <https://virusshare.com> (accessed on 20 May 2020).
33. Allix, K.; Bissyandé, T.F.; Klein, J.; Traon, Y.L. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, USA, 14–22 May 2016; pp. 468–471.
34. Google Play. Available online: <https://play.google.com/store> (accessed on 20 May 2020).
35. Fawcett, T. An introduction to ROC analysis. *Pattern Recognit. Lett.* **2006**, *27*, 861–874. [[CrossRef](#)]
36. Idrees, F.; Rajarajan, M.; Conti, M.; Rahulamathavan, R.; Chen, T. PIndroid: A novel Android malware detection system using ensemble learning. *Comput. Secur.* **2017**, *68*, 36–46. [[CrossRef](#)]

37. Milosevic, N.; Dehghantanha, A.; Choo, K-K. R. Machine learning aided Android malware classification. *Comput. Electr. Eng. Int. J.* **2017**, *61*, 266–274. [[CrossRef](#)]
38. DexClassLoader. Available online: <https://developer.android.com/reference/dalvik/system/DexClassLoader> (accessed on 20 May 2020).
39. java.net.Socket. Available online: <https://developer.android.com/reference/java/net/Socket> (accessed on 20 May 2020).
40. Orange. Available online: <https://orange.biolab.si/> (accessed on 20 May 2020).
41. Karbab, E.B.; Debbabi, M.; Derhab, A.; Mouheb, D. MalDozer: Automatic framework for android malware detection using deep learning. *Digit. Investig.* **2018**, *24*, S48–S59. [[CrossRef](#)]
42. Noreen, E. *Computer-Intensive Methods for Testing Hypotheses: An Introduction*; Wiley: New York, NY, USA, 1989.
43. Allix, K.; Bissyande, T.F.; Jerome, Q.; Klein, J.; State, R.; Le Traon, Y. Empirical assessment of machine learning-based malware detectors for Android. *Empir. Softw. Eng.* **2016**, *21*, 183–211. [[CrossRef](#)]
44. Android Enterprise Security Whitepaper. 2018. Available online: https://source.android.com/security/reports/Google_Android_Enterprise_Security_Whitepaper_2018.pdf (accessed on 20 May 2020).
45. SELinux. Available online: <https://source.android.com/security/selinux> (accessed on 20 May 2020).
46. Seccomp BPF. Available online: https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html (accessed on 20 May 2020).
47. Seccomp Filter. Available online: <https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html> (accessed on 20 May 2020).



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).