Hindawi
Security and Communication Networks
Volume 2018, Article ID 4672072, 15 pages
https://doi.org/10.1155/2018/4672072

Research Article

Analysis and Evaluation of SafeDroid v2.0, a Framework for Detecting Malicious Android Applications

Marios Argyriou,[1] **Nicola Dragoni**,[1,2] **and Angelo Spognardi**[3]

[1]*DTU Compute, Technical University of Denmark, Denmark*
[2]*Centre for Applied Autonomous Sensor Systems, Örebro University, Sweden*
[3]*Dipartimento Informatica, Sapienza Università di Roma, Italy*

Correspondence should be addressed to Angelo Spognardi; spognardi@di.uniroma1.it

Received 16 March 2018; Revised 2 July 2018; Accepted 12 August 2018; Published 5 September 2018

Academic Editor: Karl Andersson

Android smartphones have become a vital component of the daily routine of millions of people, running a plethora of applications available in the official and alternative marketplaces. Although there are many security mechanisms to scan and filter malicious applications, malware is still able to reach the devices of many end-users. In this paper, we introduce the SafeDroid v2.0 framework, that is a flexible, robust, and versatile open-source solution for statically analysing Android applications, based on machine learning techniques. The main goal of our work, besides the automated production of fully sufficient prediction and classification models in terms of maximum accuracy scores and minimum negative errors, is to offer an out-of-the-box framework that can be employed by the Android security researchers to efficiently experiment to find effective solutions: the SafeDroid v2.0 framework makes it possible to test many different combinations of machine learning classifiers, with a high degree of freedom and flexibility in the choice of features to consider, such as dataset balance and dataset selection. The framework also provides a server, for generating experiment reports, and an Android application, for the verification of the produced models in real-life scenarios. An extensive campaign of experiments is also presented to show how it is possible to efficiently find competitive solutions: the results of our experiments confirm that SafeDroid v2.0 can reach very good performances, even with highly unbalanced dataset inputs and always with a very limited overhead.

1. Introduction

The breakthrough of smartphones occurred some years ago thanks to the combination of telephony and Internet in a single portable machine, offered in many flavors, like Android, iOS, Windows Phone, and Blackberry. Nowadays, Android is by far the dominant platform, with an average worldwide market share of 85% [1]. It is acknowledged that the number of Android-based devices skyrocketed due to the open-source approach, as opposite to the closed approach followed by its competitors. That means high availability of applications outside the official marketplaces (Google Play) that Android users can freely download and install in their devices [2]. Beside being a great source of revenue for device manufacturers and app developers, availability of the apps also represent a gold opportunity for malicious developers that can hide and include harmful pieces of code (i.e., malware) in their apps [3], in order to evade the security checks of the official markets [4].

To face the challenge of Android malware detection, researchers have invested many efforts in designing systems and mechanisms to detect malicious activities in Android apps, like critical data leakages or unintentional hidden functions. The proposed solutions range within many different techniques, starting from signature-based detection, to artificial intelligence, leveraging static, dynamic, and hybrid analysis [5]. As further explored in the following sections, the most adopted approach among the detection proposals is the use of machine learning classifiers, while the most distinguishing element is the choice of the features to be considered for the classification that vary from app requested permissions and system commands to leery API calls. A

promising and also effective solution considers performing static analysis of the app binary code and relying on their different usage of API calls and packages to classify malicious and benign apps. With this direction in mind, we aim to advance an effective proposal for Android malware detection.

Contribution and Outline of the Paper. In this paper, we present SafeDroid v2.0 ([6] is a preliminary version of this paper), a complete framework to discriminate benign applications from malicious ones. A collection of scripts, a database, a web server, and an Android application form the SafeDroid v2.0 framework. These components pull together firmly in order to analyse, classify, and, eventually, decide whether an Android application is of benign or malicious nature.

The core of the framework is constituted by machine learning classifiers that are the base of real-time malware detection service for Android devices. Our solution introduces a complete, automated system for static analysis of Android applications, data extraction and storage, and comparison of different machine learning classifiers and algorithms, as well as visualisation of the produced results. The main target audiences of this framework are both the security research community and individual Android users. Our framework can be employed by Android malware analysts as an out-of-the-box solution to analyse and investigate the structural elements of malware. To foster further research activity, our framework supports many hyper-tuning parameters, which are evaluated throughout the detection process, that can be used to further explore the solution space and produce state-of-the-art and specialised models. We will show that the framework is fast, scalable, easy to use, and able to produce results of high accuracy, since it has been evaluated during an extensive campaign of experiments.

Our SafeDroid v2.0 framework is an elegant open-source solution that can also be used to provide a detailed overview and comparison of different classification algorithms and finally choose the best available model. The main novelty of SafeDroid v2.0 is the high level of automation of the testing procedure that includes application analysis, statistical analysis, feature vector extraction, service update, and feedback presentation to the user. This allows the security experts to explore many different choice of settings and options easily and extensively: the framework, in fact, supports user parameterization in the sense of allowing the user to choose exactly the conditions that the dataset has to fulfil. Eventually, the produced models can be tested in an integrated back-end server that comes along a light-weight Android application.

The rest of the paper is organised as follows. Section 2 presents related researches on the topic of Android malware detection. Section 3 illustrates the architecture of the proposed framework and a detailed overview of our methodology. We evaluate the performance of our proposal in Section 4 and we finally conclude the paper in Section 5.

2. Related Work

There are three main approaches that try to tackle the problem of malware detection. We classify the approaches based on the detection techniques they employ. We can have the static approach, which analyses the code of an application, the dynamic approach, which inspects the app behaviour at run time, and, finally, the hybrid approach, namely, a combination of the two previous methodologies. We classify and discuss the academic literature based on such three categories.

2.1. Detection Based on Static Analysis. Fuchs *et al.* [7] followed a modular analysis approach to discriminate malicious applications from benign ones. They created a tool called `ScAndroid` which allowed incremental checking of application as they are installed on the device. This tool solely extracts security specification from manifests that come along the applications and checks whether data flows are consistent with a set of predefined specifications. It uses static analysis to extract the appropriate information and makes automated security decisions based on the data flows such as deciding if it is safe to grant certain permissions to an application. Alternatively, it can pass security control to the user by providing relevant context like issued certificates and offline reviews.

Aafer *et al.* in [8] and Xiangyu *et al.* in [15] performed frequency analysis technique of the extracted API calls. In [8], the feature set was refined to use only API calls invoked by third-party applications. The APIs came mostly out of advertisement, web tracking, and web analysis packages. As the next step, they clustered the APIs by invocation methods (application specific, third-party package, or both). Their dataset consisted of 20,000 applications, 19.9% of which were malicious, and they were able to achieve 97.8% accuracy using machine learning algorithms. As a conclusion, they present statistics to argue in favour of the superiority of the API-based performance as compared to permission models. The same motif was engaged in [15] but produced opposite results. In this paper, the permission-based model managed to achieve 84.4% accuracy with the help of machine learning classifiers in contrast with the 70% that was achieved with the use of API-based model.

Ali-Gombe *et al.* in [9] followed the same mind-set. They concentrated efforts on op-code sequence analysis. After the Feature Extraction phase, the signature extraction produced a three-level cluster containing API calls made within the application, API call sequences from method signatures, and finally a collection of all method signatures for each class separately. F-Score, recall, and precision metrics were used to evaluate their model. Although they managed to get decent values (97.5%, 98%, and 97%, respectively) they argue that all their techniques could be circumvented with simple obfuscation techniques.

In another relevant publication of Bhattacharya *et al.* [10], the researchers focused on feature elimination techniques that would allow them to remove irrelevant features and thus limit the length of the feature vector set. The collection of the features is extracted from the Android Manifest file as they consider only the permissions. The feature elimination is carried out to define a subset of significant installation-time features to use in model construction. The main goal of this phase is to find a minimum set of features such that the

resulting probability distribution of data class is as close as possible to the original distribution created with all features. The limited size of the explored database (100 benign and 70 malicious) allowed the creation of 4 databases with different number of features, 83, 10, 5, and 2. K-fold validation (k=10) produced an accuracy score of 77%.

In [11], published in 2016, Yang *et al.* employed a new approach to analyse Android malware, namely, static analysis with intensive feature engineering. The authors consider features from different sources and different levels of the application. They create 5 feature sets. 3 of the feature sets come from the static analysis of the Dalvik Executable of the app, from the binary, the assembly, and the API levels of abstraction. The remaining 2 are acquired by analysing the Android Manifest file over the binary and information levels. Finally, the 5 feature sets are combined to form the final feature vector that is used for the training of the machine learning algorithm. This approach was evaluated over a dataset of 1,100 apps, out of which 50% was malicious. The research produced 98.1% detection accuracy and 8% of false positive rate, using the Random Forest classifier for a number of 50 trees.

In 2017, Fereidooni *et al.* proposed `ANASTASIA` [12], a system to detect malicious applications through statically analysing applications' behaviour. Although the techniques did not differ much from the previously discussed ones, they had the advantage of obtaining a big dataset of 11,187 malicious and 18,677 benign applications. For the model training procedure, they used 9 different classification algorithms, namely, XGboost, Adaboost, RandomForest, SVM, KNearestNeighbors, Logistic Regression, Naive Bayes, Decision Tree, and Deep Learning classifiers. The produced feature vector included 560 items and consisted of intents, permissions, system commands for root exploits, API calls, and malicious activities. They managed to perform fine tuning of the classification hyperparameters and thus achieve 97.3% accuracy on their training set.

In another work of 2017, Martín *et al.* in [13] developed a string-based malware detection mechanism employing supervised learning. Their study considered application binary information and permission-based features to estimate the nature of Android apps. The authors discuss the advantage of the string-based model over traditional signature-based ones, stating that it can deal with the different malware without the need to disassemble or run processes. To evaluate their solution, the authors compared the accuracy results with common antivirus engines. They employed the Bagging methodology to train the model and achieved a 97% discrimination accuracy, a result better than the 83% that the best antivirus engine had to offer.

The research of Milosevic *et al.* [14], published in 2017, utilised both classification and clustering techniques to discriminate between malware and benign apps in a tool called `Seraphimdroid`. The researchers formed a feature set that consisted of permissions and API calls. Android apps were first decompiled and then a text mining classification based on bag-of-words technique was used to train the model. In order to evaluate the performance of the Seraphimdroid tool, the researchers conducted experiments for permission-based and code-based clustering and classification. The results for the permission-based approach showed an F1 score of 87.1% and 64.6% correctly clustered instances while the results for the source-based approach produced maximum F1 accuracy of 95.8% and 82.3% correctly clustered instances.

In 2018, Wang *et al.* [16] extracted app characteristics found both in string values and structural features. Requested permission and API calls are acquired through the string values while function call nodes are inferred from the structural features. As a proof of point, experiments were performed on a data set of 2682 Android applications, out of which 1296 were malicious. The final results were obtained by weighting the respective detection results of the two types of features, achieving 97.9% accuracy in terms of F1 score.

2.2. Detection Based on Dynamic Analysis. Kuester *et al.* in [17] created a tool to monitor and collect relevant system events on the users' devices, namely, `MonitorMe`. The formality was to represent malware behaviour as actions while each set of actions represents a word. Collected words eventually will form a pseudo language. Every captured system call of each application is assigned a label, based on the component number and the intention of the action itself, and is sent back to the server responsible for determining the nature of the applications. Additionally, MonitorMe framework defines policies against privilege escalation attacks which aim on gaining elevated access to protected resources. The tool recognises malicious payload launches that occur after system boot. Among the drawbacks of this approach was that the device needs to be rooted throughout the procedure and the high percentage of false positives against true positive values, at 28% and 93.9%, respectively.

Burguera *et al.* in [18] try to challenge the obfuscation problem. They developed a crowdsourcing framework to collect samples of execution traces of applications, namely, `CrowdDroid`. Its main functionality lies on monitoring the underlying Linux kernel system calls and reporting the findings back to a centralised server. The primary goal is to identify different behaviours in applications that share identical names. The server then creates a dataset of behavioural data for each application. They cluster each dataset, using partitioning clustering algorithms, to differentiate between benign applications that demonstrate similar system call patterns to malicious ones.

Enck *et al.* in [19] demonstrated a tool named `TaintDroid` for real-time analysis by leveraging Android's virtualized execution environment. Their goal was to track the flow of private sensitive data through third-party applications, based on the assumption that downloaded third-party apps are unreliable. The tool monitors how these applications access and manipulate users' personal data. In order to do so, it observes which data hit the outgoing interface and their respective path by labelling them. The tool logs the data labels, the application responsible for transmission as well as the destination. The researchers observed 30 third-party applications found in Google Market and they discovered that the 65% of them transmitted sensitive data without user consent.

2.3. Detection Based on Hybrid Analysis. Sharid *et al.* in [20] focused on analysing application native code, monitoring, and detecting abnormalities during runtime. Native code refers to machine code that is executed directly by the main processor, opposed to bytecode, which is executed in the virtual environment of the Android JVM. Their static analysis focuses on defining certain patterns in the control flow. The patterns target ostensibly insignificant incidents that could pass otherwise unobserved. Their tool, `DroidNative`, is also able to evaluate bytecode by making use of the Android runtime. ART is the managed runtime that is used by applications and system services to compile bytecode and execute the Dalvik Executable format. In this case, ART is used to compile bytecode into native code for the purpose of analysis. Their experiment dataset consists of 5,490 applications, 1,758 of which are malwares. The two experiments produced contrasting results; the first one had 98.5% accuracy with a variation of 1% while the later ranged from 70% to 80%. The false positive rates showed high values for both, the lowest being around 20% and the highest up to 40%.

A case study by Poeplau *et al.* in [21] revealed that around 10% of the applications found in the official Android market are loading external code at runtime. This percentage rises further when one refers to the most popular applications. The primal intention of the research was to find commonalities among the dynamic loading of code by creating a tool to automatically detect loading attempts using static analysis techniques. The efforts were gathered around examining Java class loaders, package contexts, code disassembling, and ART. After finding that code-loading mechanisms are employed by the majority of benign applications, the authors suggested security policy reformation concerning dynamic code-loading allowance.

`DroidScreening` [22] framework scans applications at runtime to identify malicious patterns of execution. The solution followed the standard objectives of the antivirus industry, that are the malware sample screening and the identification of threat level. `DroidScreening` employed both static analysis to extract key identifiers of the targeted Android application to train the machine learning model and lazy associative classification algorithms to produce the classification model. The novelty of the solution lays on the trigger dynamic execution-based analysis environment that generates various system-wide events to trigger malicious activity. Their dataset numbered 1554 malware and 2446 benign applications. `DroidScreening` scored an F1 value of 91.1%. In addition to F1 score, the authors crafted a metric for the average misclassification cost, because they state that the cost of different misclassification errors should not have the same effect on the final accuracy result.

The idea behind the work of Bae *et al.* [23] was a detection system that monitors network usage, network connections, APIs, and permissions of an Android application. The first two were observed dynamically at run time while the latter were statically extracted by the application. The novelty behind this idea was to tackle the advent of new types of malware threats rather than focusing only on the already identified families of malware. The choice of building a hybrid solution is defined by observing that dynamic analysis adds significant amount of overhead to the device and static methods are easily evaded by obfuscation. The authors complimented the two methods to maximise the detection accuracy and minimise the introduction of high overhead functions. SVM classifier was employed to produce the machine learning model. The evaluation tests, conducted on a dataset of 413 benign and 398 malicious applications, produced 90.3% precision rate and a false positive rate of 13.9%. Moreover, the solution introduced a maximum overhead of 22.7% at the Android device.

2.4. Limitations of the Existing Approaches. The detection based on static analysis does not consume many resources and produces the most accurate results. From the negative side, there are techniques that can obfuscate the results, such as code encryption and dynamic load of code. Dynamic analysis techniques are more sophisticated but add a serious amount of computation overhead. They surpass static analysis in being resilient to evasion, code obfuscation, and polymorphism. They run as daemons processes, constantly monitoring the device for abnormal execution patterns or unprivileged information flow. Thus, they consume more resources, in terms of power and bandwidth, which leads to the degradation of the user experience. The observation of vague patterns produces high false positive rates, which makes dynamic analysis unfeasible in real-time scenarios. While it seems the most promising approach for future detection mechanisms, hybrid analysis is still in premature stages of development.

Finally, we can see how all the proposed solutions are not easily customisable, but are always provided as final solutions. Given the dynamic and rapidly changing field of malware creation, it is desirable to have an automated framework that allows to experiment and test with many different combinations of machine learning classifiers, with a high degree of freedom and flexibility in the choice of features to consider, dataset balance and selection. This can also give any attentive researcher the possibility of experimenting and validating unsolicited ideas and not common combinations of parameters.

3. SafeDroid v2.0 Framework Architecture

In this section, we present the architecture of our framework, namely, how the components are related as well as the logical steps of the execution. To enforce flexibility in the framework, each component is designed to run independently from the others. This means that each one has its own self-standing functionality and that each step can be iterated, split, altered in any component, and fully customised, according to the parameters defined by the user.

Figure 1 presents an overall overview of SafeDroid v2.0, with the main components and how they logically depend on each other. The framework can be summarised in one Android app, responsible to interact with the user, five main components, and a database. The first component that operates on the original dataset is the Feature Extraction

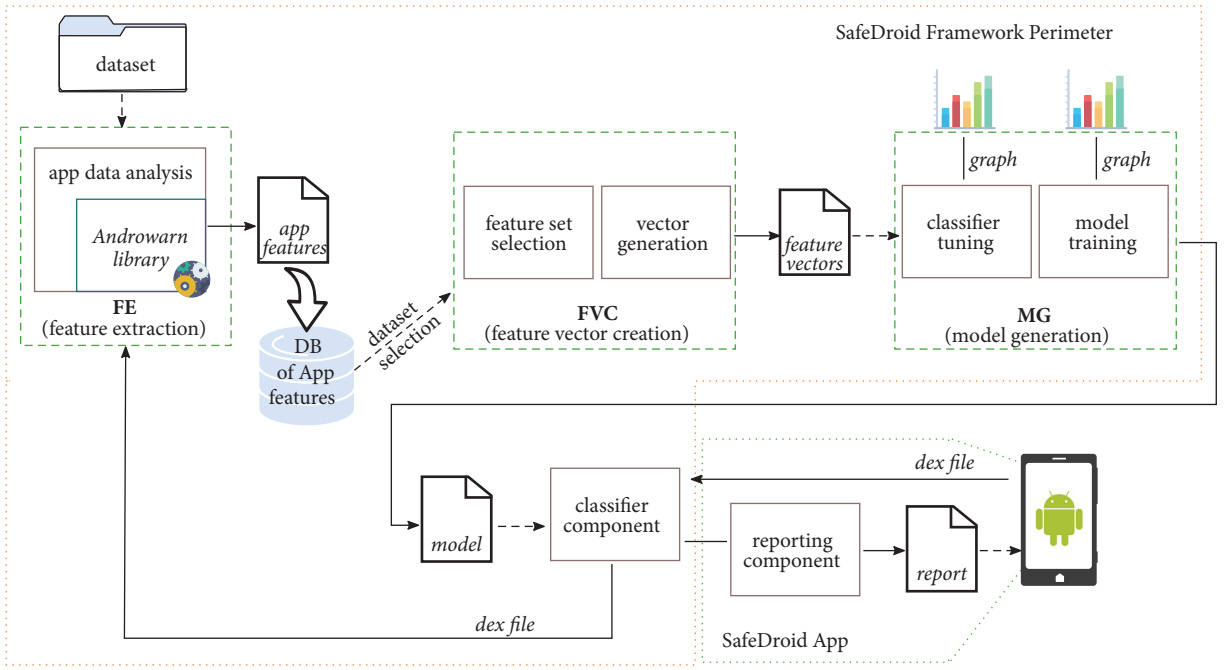


Figure 1: Overview of the SafeDroid v2.0 framework architecture, namely, the components and their interaction.

component that is responsible to reverse engineer the application binary code and to store the output in a database. The Feature Vector Creation component, which comes next, determines the most suitable features to use and builds the training and the testing sets for the Model Generation component. Several classifiers are generated and the most accurate one is selected as the model to be adopted in the SafeDroid v2.0 app. Finally, the model is exported to the Classification component that sends the final result of the analysis to the Reporting component that is the main interfaces with the Android users.

Before detailing the SafeDroid v2.0 framework, it is important to make an important premise about how the user privacy is taken into account when working with sensitive data as users' data and users' application preferences. SafeDroid v2.0 is a tool as many others proposed in literature: since the main focus of the security community is to protect against malware, the collection of user data and application list is an established practice in this research field. Thus, we can observe that we think that is important to protect the privacy of the users, but probably it is more important to protect her data against the malicious activity of malware (and then using all the possible resources for this aim).

As academic researchers, we provide as an additional guarantee of our *bona fide* several suggestions on how to address the privacy issue. First of all, we already released the full source code of our framework (https://github.com/Dubniak/SafeDroid-v2.0), together with the datasets and the parameters we used to perform the experiments presented in this paper: this means that it can be used directly by the Android user itself, without the need to give away its data to anyone and, still, check for the safety of its running or ready-to-install apps.

Secondly, we can easily imagine to include a transparent "anonymiser" element that addresses all those elements that could be used to mine the privacy of the user: the literature about this form of data-processing is huge and still growing, as an additional proof that the overall objective to obtain the security of the users is of outmost importance and can be pursued adopting the right tools that the state of the art offers.

In the following, we detail each single component and describe its main functionality together with the interaction among each component.

3.1. Feature Extraction. The Feature Extraction (FE) component is responsible for the analysis of the input apps. The main source of the apps for building the knowledge of the system is a dataset of labelled apps (malicious or benign). However, after the initialisation of the system, new apps are provided by the user for analysis and detection purposes, by means of the SafeDroid v2.0 app.

As a static-analysis-based framework, the role of the FE component is to extract all the useful data from the binary code of a given app such as identification information (name, alias, date, check-sums, etc.) and primitives that highlight the use of the app (API, permissions, third packages, etc.). The app analysis is performed by the FE component employing the functionality of `AndroWarn` (https://github.com/maaaaz/androwarn). AndroWarn is an open-source tool for reverse engineering of Android applications. The Feature Extraction component leverages an optimised revision of the library to perform a static analysis of the Dalvik bytecode of an application, which is represented as a Smali (intermediate file format between Dalvik Executable and Java source code). As detailed in Figure 2, the user has to provide as input a dataset of applications that will be

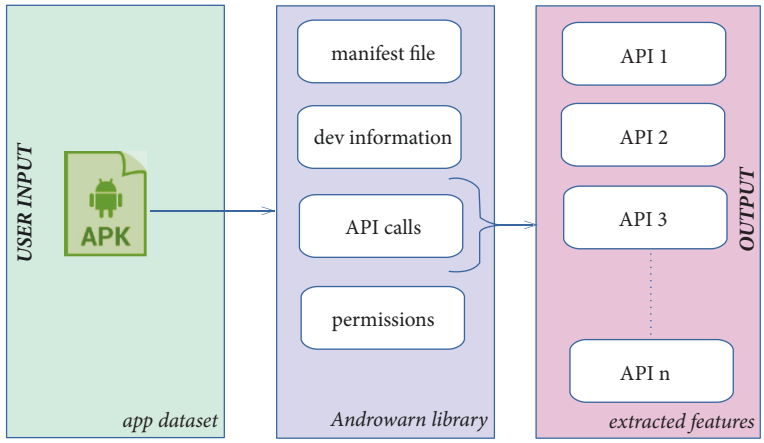


Figure 2: Overview of the Feature Extraction process.

processed by the FE component. Each app is analysed by the optimised AndroWarn library and all the APIs are extracted from the bytecode, to become the extracted features.

Since the app analysis is generally time and resource expensive (as detailed in Section 4), the result of the reverse engineering is permanently stored in another component of SafeDroid v2.0, the database, together with a label that designates the nature of an app. In this way the framework can realise a large dataset of applications that can be continually expanded, as new apps arrive at the FE component.

As in any other machine learning solution, the dataset constitutes the knowledge base for the whole framework, from the training to the testing phase up to the detection mechanism and all the other SafeDroid v2.0 components rely on it to perform their functions: for example, the Feature Vector Creation generates the input for the Model Generator (namely, feature vectors) evaluating the extracted features and their usage among the considered apps, while the Model Generator builds the classifiers leveraging the feature vectors. The distinction between malicious and benign applications managed by the FE component is critical for the correct outcome of the whole detection process.

3.2. Feature Vector Creation. The Feature Vector Creation component (Figure 1, shortened to FVC from now on) performs three main functions: it extracts the features from the dataset of apps (accessing to the DB), then it finds the most discriminatory features the detection mechanism will later use to identify the nature of any app, and, finally, it generates a feature vector for each app that will be the input for the next components. An overall picture of the involved steps is depicted in Figure 3. The FVC is tailored in such way to allow the user to choose the desired execution parameters, such as the size and the balance of the dataset: the high efficiency of the whole evaluation process allows to build several experiments that can be used to empirically evaluate and compare several possible solutions. Since the main target of SafeDroid v2.0 is the security research community, we give freedom to the user by providing a framework with many parameters that supports unconstrained execution modes. For example, one execution could be set to consider only a small part of the dataset, another one to consider only the dominant malwares, i.e., those apps that have the most distinguishable behaviour. As soon as the app dataset is selected and the parameters for the analysis are provided, the SafeDroid v2.0 framework is able to automate all the following steps, up to the generation of the model. However, this setup is, eventually, one of the most critical steps. Thus, users should acknowledge potential biases to the results that are induced by the choice of the execution parameters. However, as many other frameworks publicly released, our tool can only help the experts exploring several (possibly optimal) solutions, but we can clearly miss the *best* solution, as part of the human bias. In some sense, we could say that we can help removing the human bias, providing a framework that makes testing in an automated way several settings in parallel possible, but we can not eventually exclude a human bias or mistake in principle.

For this reason, the SafeDroid v2.0 allows the creation of multiple datasets in order to prepare the knowledge base for different experiments. Each dataset will be possibly composed of different features (namely, APIs), such that it will be able to generate different app classifiers. The whole procedure can be iteratively repeated when new applications are added to the whole dataset, to generate updated versions of the feature sets that, in turn, will generate new, updated versions of the app classifiers.

We will now detail all the steps performed by the FVC component in order to generate the feature vectors for the apps in the dataset. We start considering the dataset D the user provides for the Feature Selection phase.

Given a dataset of applications D, we can see it as the union of two distinct subsets, namely, $D = B \cup M$, where B and M are the subset of the *B*enign and the subset of the *M*alicious applications, respectively. The idea is to build a feature set leveraging the frequency of each API in every app in the dataset. The frequency is deducted by dividing the occurrences of each API over the total occurrences that the

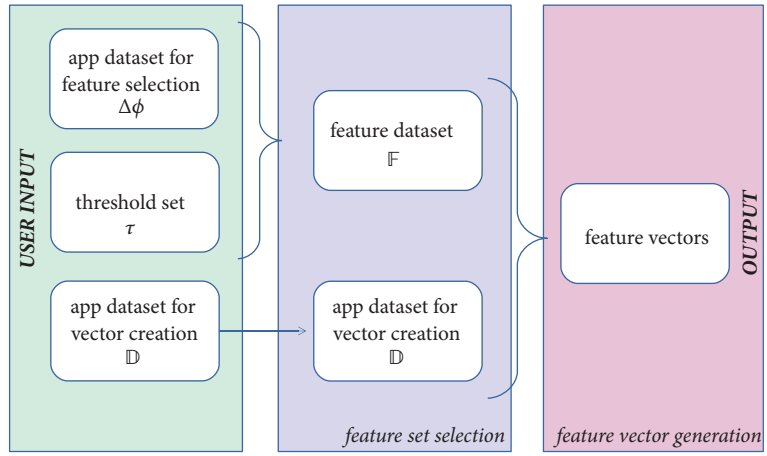


Figure 3: Overview of the Feature Vector Creation processing steps.

specific API has in the dataset, for both the malicious and the benign ones.

More formally, if we denote as App_X ⟵ API.a the fact that application App_X makes use of the API API.a, we can define a sort of indicator function (from Wikipedia: an *indicator function* is a function defined on a set X that indicates the membership of an element in a subset A of X, having the value 1 for all elements of A and the value 0 for all elements of X not in A) $\mathbf{1}_X(a)$ as

$$\mathbf{1}_X(a) = \begin{cases} 1 & \text{if App\_X} \longleftarrow \text{API.}a, \\ 0 & \text{else} \end{cases} \quad (1)$$

and, then, we can evaluate the frequency of API.a in the subset of the benign applications ϕ_a^B as

$$\phi^B(a) = \frac{1}{b} \sum_{X \in B} \mathbf{1}_X(a) \quad (2)$$

where b is the number of benign applications, namely, $b = |B|$. We can similarly define

$$\phi^M(a) = \frac{1}{m} \sum_{X \in M} \mathbf{1}_X(a) \quad (3)$$

as the frequency of API.a in the subset of the malicious applications, where m is the number of malicious applications, namely, $m = |M|$.

With the two values $\phi^B(a)$ and $\phi^M(a)$ we can, then, evaluate the difference

$$\Delta\phi(a) = \left|\phi^B(a) - \phi^M(a)\right| \quad (4)$$

that denotes if a given API.a is more frequent in one of the two subsets: informally, we can tell that the higher its $\Delta\phi$, the higher the power of an API to discriminate as malign or benign, an application that uses such API.

To perform the feature set selection, we set a threshold t for the value of $\Delta\phi$ and we consider only those API a such that $\Delta\phi(a) \geq t$. In this way, we are actually selecting a subset of the whole set of the APIs, with a discriminating power that depends on t. Such a subset of APIs can be used as a feature set to build a classifier for the dataset of our applications. Furthermore, if we select more than one threshold, we will have more feature sets. SafeDroid v2.0, indeed, given in input a set of k thresholds $\tau = \{t_1, \ldots, t_k\}$, is able to generate k different feature sets that can be used for the next steps of the classifier generation. In particular, given a set of thresholds $\tau = \{t_1, \ldots, t_k\}$ as input from the user, the framework generates a family $\mathbb{F}$ of feature sets as

$$\mathbb{F} = \bigcup_{t \in \tau} \{a \mid a \longleftarrow \text{App\_X} \in D \text{ and } \Delta\phi(a) >= t\}; \quad (5)$$

that is, for each threshold t in τ, it generates a new set with those APIs that belong to any of the apps in D, such that the $\Delta\phi$ of those API is greater than or equal to the threshold t. Each of those new sets is included in the family of feature sets $\mathbb{F}$. We can informally say that, for each value of a threshold, the framework resolves a feature set, namely, a set of APIs that tries to describe the behaviour of an application.

The last operation of the FVC is the generation of the relevant binary vector for a given app, namely, a representation of an app suitable to input the machine learning algorithms and generate a classifying model. The idea is to have a vector encoding the use of each API of the feature set for every app: each API is assigned a dimension of the vector, where there is a 1 if the corresponding API in the feature set is used by the app, and 0 otherwise. For this purpose, the FVC considers each app in the dataset, one by one, and checks if it uses any of the APIs of the selected feature set, again one by one.

For example, suppose to have an application App_X which uses the set {API.a, API.c, API.d, API.e, API.f} of APIs. Given the feature set {API.a, API.b, API.c, API.g}, we would obtain a binary feature vector

$$\text{FV(App\_X)} = 1\ 0\ 1\ 0 \quad (6)$$

In this example, the first dimension is 1, since it corresponds to API.a and this API is used by App_X, the second dimension is 0 because API.b is not used by App_X, and so

on. The number of the produced feature vectors depends on the number of apps in the dataset provided in input and the number of thresholds in τ, while their length will depend on the feature sets in $\mathbb{F}$.

Besides the option to receive a static dataset as input, SafeDroid v2.0 helps the setup of a batch of experiments since it is able to randomly generate a family of app datasets, with different combinations of parameters: namely, the user can specify any combination of dataset sizes (s) and balance (r) between malicious and benign applications, and the framework will generate datasets with the given characteristics. More formally, given a set ρ of pairs (s, r), where s is the expected size of the dataset and r is the expected ratio of the malicious applications in the dataset, the framework will produce a family $\mathbb{D}$ of app datasets as

$$\mathbb{D} = \bigcup_{(s,r)\in\rho} A_{s,r} \tag{7}$$

where, for each $A_{s,r} \in \mathbb{D}$, we have that

$$\left|A_{s,r}\right| = s, \tag{8}$$

$$\frac{\left|A_{s,r} \cap M\right|}{\left|A_{s,r}\right|} = r \tag{9}$$

Each dataset in $\mathbb{D}$ will be used as input for all the phases (Feature Selection, Feature Vector Creation, and the following steps described next) and will produce different results and classifiers. We just recall that the generation of $\mathbb{D}$ is very efficient because it is produced leveraging the application database, built in the previous Feature Extraction phase.

3.3. Model Generation. The goal of the Model Generation component (Figure 1, MG from now on) is to choose the best classification method for a given dataset of (D) received in input and to build the relative classifier (trained model) [24]. This is represented in Figure 1 by the *classifier tuning* and *model training* steps, since the MG evaluates several different machine learning classifiers and their most effective classification tuning parameters to maximise the accuracy metrics and minimise the mis-classifications. Again, we stress that the choice of the machine learning classifiers and the parameters are full responsibility of the user, since SafeDroid v2.0 is a framework to support the analysis of several possible settings of options and parameters.

The MG component supports three modes of execution, namely, `optimum feature vector`, `optimum dataset`, and `unattended`. In the optimum dataset mode, the dataset is split in subsets, as defined by the user. The framework then trains the prediction model based on the optimum slice of the dataset that is the one that provides the highest accuracy and lowest negative results. In the optimum feature vector mode, the framework creates multiple feature sets based on the user choice. Then, it adopts the one that optimally describes the nature of the apps. It is crucial to notice that the final accuracy results are obtained by testing on the whole dataset. In the unattended mode of execution, the framework does not apply any optimisation algorithm.

The methodology adopted for SafeDroid v2.0 is the standard machine learning trial-and-error approach [24] and the selection is done comparing the outcome of two procedures: a standard cross-validation experiment and a greedy look-up with a split of the input dataset in training/testing subsets. The parameters for both the procedures are set by the user (number of folds for the cross-validation and the training/testing set ratio).

In our experiments, the SafeDroid v2.0 framework employs the functionality of the following classification algorithms: K-Nearest Neighbor, Support Vector Machine, Random Forest, Decision Tree, Multilayer Perceptron, and Adaptive Boosting [25]. In particular, the framework uses the cross-validation technique to produce accuracy results of all classifiers and then compares the cross-validation scores of the different classifiers with the greedy look-up procedure. If the two values are different (e.g., the classifier algorithm), SafeDroid v2.0 reinitiates the procedure to produce one final classifier that will be used to produce the prediction model. At this stage, graphs that visualise both the learning and the training procedures are displayed to provide feedback to the user.

The findings of the previous phase (i.e., the classification algorithm along the tuning parameters) are loaded into the Model Generation component. During this last phase, the SafeDroid v2.0 inputs the tuned classifier to the constructor of the prediction classifier and creates the prediction model that will be employed in the classifier component.

3.4. Classification and Report Components. The Classification component (CC) works together with the Report component (RC). The CC is responsible for identifying and detecting malicious apps, while the RC task is to handle the interaction with the user. The CC receives the input data (namely, MD5,.dex) from the Android app and uses the precalculated feature set of APIs to create the binary feature vector, employing the functionality of the Feature Extraction component. The binary vector is finally tested on the prediction model and a new label is generated for the new app. This output is used to generate a report, with a feedback to be presented to the user, by means of the Android application. The report contains the features that were found to be considered malicious, so that the user can have an understandable feedback about the label given to the evaluated app. Additionally, the data extracted from the app are stored in the database component, in order to be used to update the prediction model.

3.5. SafeDroid v2.0 Android Application. The RC of the Android app retrieves the list of already installed applications on the device and sends the `dex` file of the objective application, along with the application name and the MD5 hash of the APK to the Classification and Report server. Finally, it presents the received report from the server to the user of the Android app.

Table 1: Composition of the datasets used in the experiments. The ratio column is the malicious app ratio in the considered dataset.

app dataset	size	ben. apps	mal. apps	ratio	APIs
Group 1	29344 kB	428	432	0.50	3491
Group 2	38306 kB	371	211	0.36	7007
Group 3	138613 kB	513	755	0.60	14327
Group 4	201897 kB	138	392	0.75	23229
Group 5	261358 kB	2843	755	0.20	37249
Group 6	313787 kB	1013	1611	0.60	26694
Group 7	342717 kB	1710	1611	0.50	32791
Group 8	451389 kB	3121	1611	0.33	50835
Group 9	972093 kB	3371	2481	0.40	94825

4. Evaluation

This section presents the results of our extensive experiments performed with our framework, that we recall is fully implemented and already released for further study to the research community (https://github.com/Dubniak/SafeDroid-v2.0.).

The purpose is to provide evidence about the SafeDroid v2.0 key characteristics, namely, feasibility of deployment, low computational cost, and detection accuracy. In particular, with the term feasibility we mean that the whole mechanism is actually feasible and easy to use. With low computational costs, we want to show that the whole detection process, together with the training and testing phase, is scalable and can be executed in real time. Finally, to show the detection accuracy, we want to report real use cases that confirm the capability of the SafeDroid v2.0 framework to distinguish between malicious and benign Android apps.

A total of 25,346 applications compose our *whole dataset*, out of which 20,208 (79.7%) are benign and 5,138 (20.3%) are of malicious nature. These applications were used as the knowledge base for our experiments. Part of our malware data has been kindly provided by the authors of a prior piece of research work [26]. The rest of the samples were collected from malware repositories such as `VirusShare`, `contagio Mobile`, `malware.lu` and applications dated between 2012 and 2016. The benign applications were downloaded from the official Google Play Store during the year 2013.

4.1. Feasibility of Deployment. The fully functional framework has been implemented and is available to be forked at the `github` hosting service. In particular, the version we used for the experiments in this paper has been developed with SDK v.24 on Android OS 7.0, using the `scikit-learn` and the AndroWarn library.

Given a fully functional prototype of the whole framework, we were able to carefully analyse and evaluate the single component that constitutes our implementation of SafeDroid v2.0, conducting a series of experiments. The experiments were conducted on 9 different input datasets, subsets of our initial whole dataset, over a series of 3 executions. The input datasets are detailed in Table 1 and were crafted to deliberately differ in both absolute number of apps, balance among malicious and benign apps, and number of employed APIs. Moreover, in order to test the feasibility of our framework, the settings of the execution were different in all sets of experiments. In this way we were able to acquire metrics to evaluate the computational costs of each architectural component and the efficiency of the produced prediction models on both balanced and unbalanced input datasets. We have to highlight that, since some of the experiments take a considerable amount of time, there are many external factors that influenced the framework, like the network connection to query the database, the race conditions among threads-pointers to the database, and so on. All the times we report in the following, where they are not differently stated, do not take into account such a fine grained detail, but simply report the single steps as a whole.

The experiments were conducted on a computing station with 24 CPUs (Intel® Xeon®) running at 2.76GHz with 64GB available memory.

We tested our framework on different number of processors to be able to estimate its overall performance in real-life scenarios and highlight the effect of parallelization. The base of our experiment employed 2 cores and doubled up, until to 16 cores. Table 2 presents the execution time in seconds to perform the whole Model Generation. It can be observed that the execution time decreases dramatically as the number of the employed cores increases. To grasp the overall improvement, we report the average execution time utilisation in Table 3. The results report the percentage of the enhancement of the execution time with respect to the previous setup (half the cores) and with respect to the original setup (2 cores). As the results show, as a harsh estimation, doubling the number of the cores results in decreasing the execution time by around 30%, while the overall reduction time reaches up to 65% for 16 cores.

4.2. Computational Costs. In this section we report our study to evaluate the computational costs of our SafeDroid v2.0, studying the time in seconds it takes to go through all the steps of the framework. In particular, we highlight the timing of the different phases as well as the improvement in speed we obtain using the database. To evaluate the framework in different settings, we designed a battery of tests. The experiments were conducted for 9 different input datasets (Table 1) over a series of 3 executions and they were conceived to highlight the costs of each phase of the framework and to provide evidences about the actual feasibility of the approach.

Table 2: Empirical time (in seconds) of the overall execution for each group.

app dataset	2 cores	4 cores	8 cores	16 cores
Group 1	210	112	67	38
Group 2	238	145	105	92
Group 3	1208	670	435	352
Group 4	1648	1322	1177	975
Group 5	2168	1790	1641	1314
Group 6	3396	2597	1910	1319
Group 7	4704	2577	1544	1108
Group 8	8610	4855	3293	841
Group 9	22498	15191	9820	7075

Table 3: Utilisation of execution time per core number.

number of cores	relative improvement	absolute improvement
2	-	-
4	34.7%	34.7%
8	28.5%	52.2%
16	30.4%	65.6%

To evaluate the costs of the Feature –API– Extraction phase (Section 3.1), which we recall to be based on the AndroWarn, we report in Table 4 the timing of the 3 executions for each group. The beneficial effects of the use of a database are all in the reduction of these timings. Moreover, to have an idea about how much the Feature Extraction phase impacts on the overall timing of the process, we draw in Figure 4 a plot for each of the 9 groups: they represent the timing of the single phases of the whole process, together with the overall timing, to have a grasp about how relevant such phase is in the whole process. The error bars in the Y-axis represent the variation between the slowest and the fastest execution of the several repeated experiments for each group. We can observe that, for all the groups, the Feature Extraction phase is always the predominant component for the overall execution time, being almost always close to 95% of the overall cost. Moreover, as the size of the apps dataset increases, the other phase that increases its time consumption is the Tuning one, where the framework tries different tuning parameters in order to maximise the results of the different machine learning algorithms. The Feature Vector Creation and the Training phase are always very limited in their time consumption.

We recall that the computational cost of the Feature Extraction phase is caused by the reverse engineering of the input apps: for example, in the case of an input of around 1GB (Group 9 of Table 1), it occupies most of the CPU, reaching at a peak of almost 21,000 seconds. We can also observe that the execution time of the Feature Extraction phase follows a linear increase towards the size of the input dataset.

The above results motivate our decision to include a database for permanently storing the results of the Feature Extraction for each app. In this way, the most costly operation is substituted by a simple database access, reducing by several

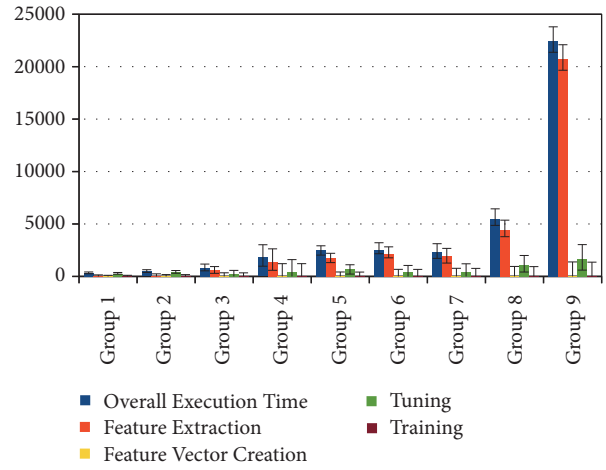


Figure 4: Comparison between execution timings, with the detailed time of each single phase. Times are in seconds.

order of magnitude the overall cost. This design choice follows the initial goal to create a framework that can be really used to assist security researchers and able to produce results quickly. We report in Table 5 the comparison between the overall time execution of the whole generation process, respectively, without and with the use of the database. In Figure 5, instead, the overall execution times are compared, again with and without the employment of the database, in order to have a visual representation of the improvement. The noticeable difference between the first—during which the reverse engineering of the input apps is taking place—and follow-up executions is visible for all sizes of the input. We can observe that as the size of the dataset to process increases, the advantage of the database increases even further: for example, for the smallest dataset (Group 1) it goes from around 5.1 minutes (316 sec.) without the database to 4.5 minutes (278 sec.) with the use of the database, with a saved time of around 11% of up to almost 92% for the largest dataset (Group 9), from 6.2 hours (22422 sec.) to 28 minutes (1734 sec.). Then, we can conclude that the adoption of the database makes extensive use of the framework—even with very large datasets—possible.

4.3. Detection Accuracy. Next, we explored the achievable detection rates of our SafeDroid v2.0 framework. The graph

Table 4: Feature Extraction timings for each group of apps. The results consider 3 runs for each group. Times are in seconds.

	Average	Min	Max	StDev
Group 1	41.59	32.85	58.14	14.34
Group 2	73.81	58.58	104.22	26.34
Group 3	597.58	301.07	922.16	311.50
Group 4	1414.33	645.07	2489.35	959.40
Group 5	1792.47	1451.24	1990.99	296.83
Group 6	2151.28	1801.20	2801.05	563.28
Group 7	1905.05	1216.81	2703.02	749.16
Group 8	4428.34	3466.56	5729.38	1168.94
Group 9	20734.27	19578.27	22275.95	1389.58

Table 5: Overall execution time comparison, without and with the use of the database. Times are in seconds.

	without DB		with DB		
	Average	StDev	Average	StDev	saved time
Group 1	316.05	91.08	278.24	76.76	11,96%
Group 2	467.76	148.72	396.30	122.41	15,28%
Group 3	837.53	324.16	245.52	12.66	70,69%
Group 4	1804.24	1075.82	391.67	120.77	78,29%
Group 5	2506.04	447.69	730.24	231.79	70,86%
Group 6	2541.09	579.78	401.92	25.46	84,18%
Group 7	2354.42	708.86	464.69	46.78	80,26%
Group 8	5507.69	827.99	1101.75	582.74	80,00%
Group 9	22442.46	1239.60	1734.36	797.11	92,27%

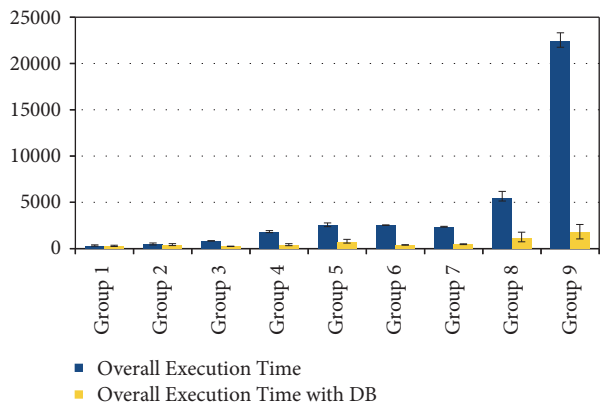


Figure 5: Comparison between the overall execution times, with and without the use of the feature database. Times are in seconds.

in Figure 6 shows the accuracy rates in terms of false positive rate, precision, specificity, and F1 values for the three different modes of execution (Section 3.3). The false positive rate is the ratio of false positive number over the total number of malicious apps in the datasets and it shows the percentage of falsely characterising benign apps as malicious ones. The specificity shows the proportion of benign malware that is correctly identified as benign and the precision refers to the framework ability of correctly identifying malware samples. The F1 score is used to measure the accuracy of the prediction model. The framework produced highly accurate results while it managed to confine the false positive rates. The results are in adequate level, acquiring values up to 99.5% for the positive indices. It is interesting to observe the framework behaviour when the input dataset is the group 4. As we recall from Table 1, group 4 refers to the highest unbalanced dataset, which consists of 75% malicious and 25% benign apps. In this case, the prediction model identified almost 93% of the cases while the false positive rate climbed up to almost 12.5%. Overall, the framework scored the highest accuracy for the optimum feature vector mode of execution.

The ability of the framework to choose the most accurate classification algorithm among the 6 available candidates can be observed in Figure 7. For the purpose of this experiment, we recorded the results for the 9 input datasets in terms of F1 score. The red colour denotes the framework prediction for the classification algorithm with the highest accuracy. The framework successfully chose the optimum classifier in all experiments, allowing the optimum alternative for the prediction model. As discussed in Section 3.3, the choice of the algorithm is taken by comparing the cross-validation scores that are produced after the exhaustive search of the tuning parameters for each candidate. The final scores, which are displayed in Figure 7, refer to the prediction models that are exported after the training and testing procedure (Figure 1). As so, the results provide a strong insight on the versatility of the fine tuning module; the vast majority of the prediction model scored values above 95%.

In order to observe the ability of the framework to produce prediction models on big input dataset, we conducted 3 experiments on the full population of the dataset that

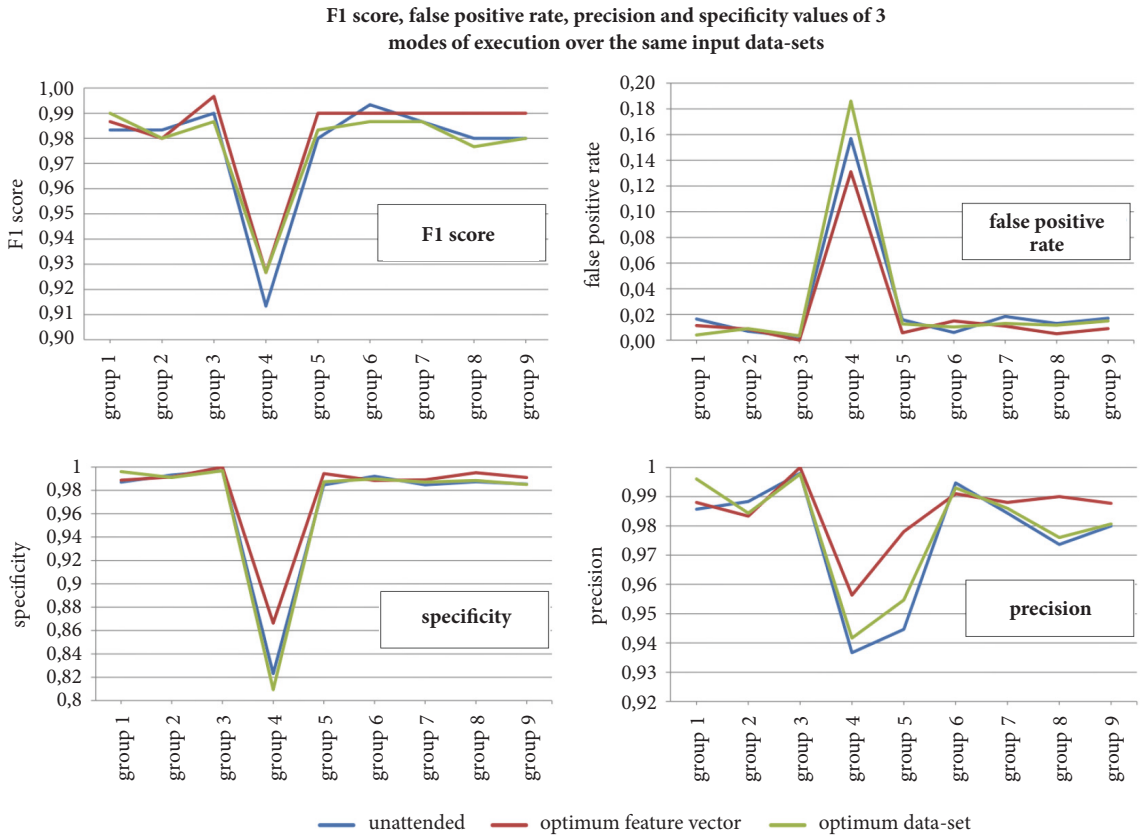


Figure 6: Accuracy scores for three different modes of execution: unattended, optimum feature vector, and optimum dataset.

we obtain (25346 apps, 20.3% malicious), for a series of 3 executions. The base hypothesis of our experiment is that the framework can produce acceptable prediction rates for both limited and extended feature sets. The choice of the feature is the key to characterise an app as malicious or benign. For the purpose of this experiment, the framework created 3 distinctive feature sets, subgroups of the initial set. The length of the feature sets along the prediction rates is presented in Table 6. The available feature space for the whole dataset is around 1.4 billion unique features. In the 3 consecutive cases, the framework relied on 157, 1,879, and 2,428 features, respectively, to construct the prediction models. The average F1 accuracy ranges from 96.9% for 157 features to 98.9% for 2428 features while the false positive rate is constrained in between 0.6% and 1.4%, respectively. Moreover, the punctual choice of the most discriminatory features is inferred from the results; the framework produced results of high accuracy for a feature set consisting of 157 characteristics of malicious apps. While the feature set expands, the framework gives higher accuracy values and lower false positives.

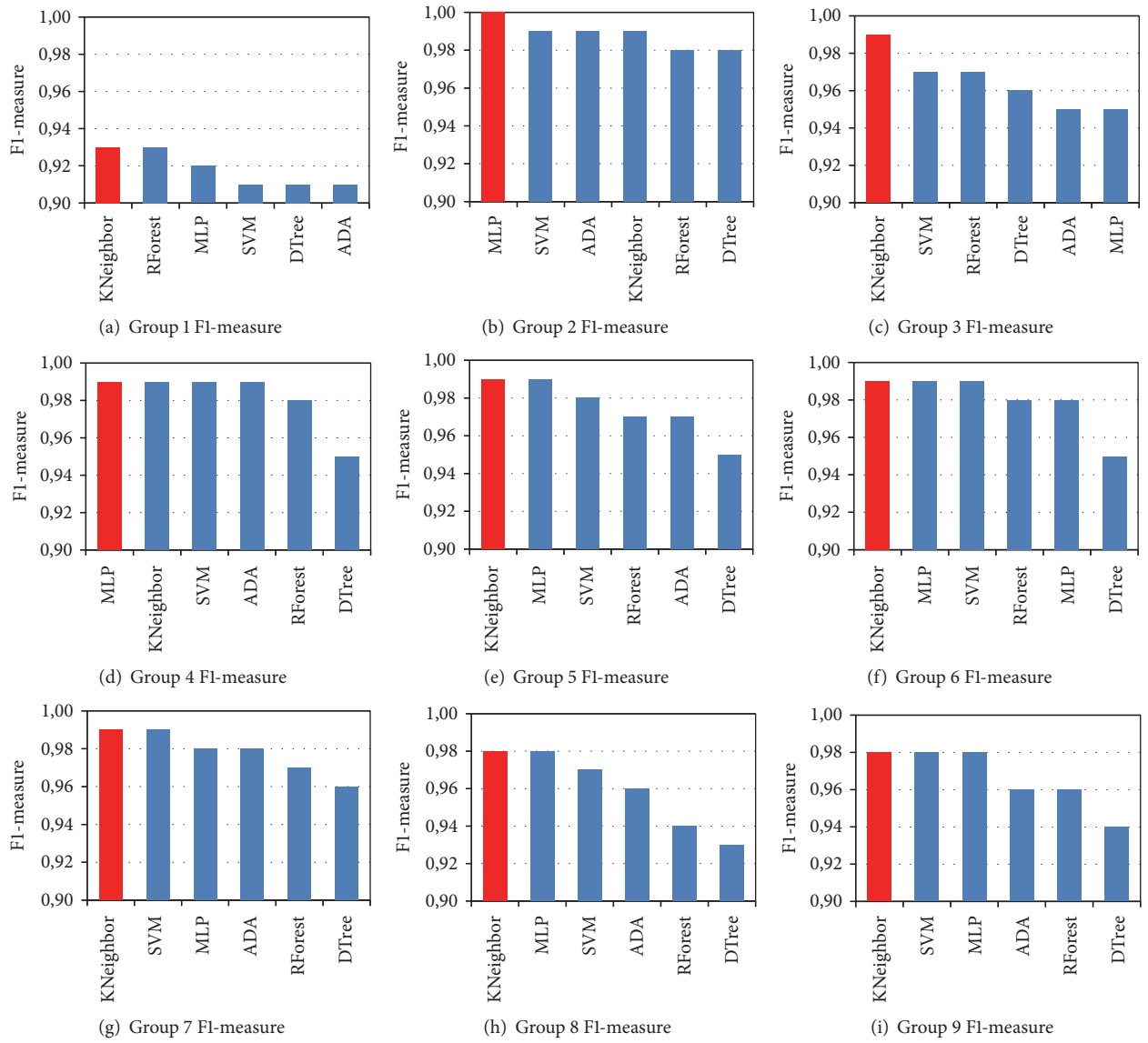
4.4. Performance Comparison. Table 7 compares our proposed solution with the most prominent alternatives as the state of the art of the literature. All compared frameworks follow the static analysis approach to produce their results. The table displays the F1 and False Positive Rate, which are the most common metrics to measure the performance of a framework, the number of the employed machine learning algorithms, and, finally, the ability of the framework to be employed for further security research. The dataset column refers to the population of the examined applications, while the malicious column refers to the proportion of malicious applications within the dataset. We use these two columns to point out the dataset imbalance in the compared methods. As we observed, a lower proportion of malware in the dataset results in a higher F1 accuracy and lower false positive rate. Feature column introduces the attributes that were used to examine the nature of the applications. All of the aforementioned metrics were employed by most of the related researchers and have been evaluated as the best candidates to measure the real performance of the solutions.

In order to compare the other solutions with our framework, we used the prediction scores that were produced by using the full dataset with the highest dimensionality of the feature space (Table 6). Although the number of malicious applications in our dataset is slightly lower than the average

Table 6: Summary for 25,346 apps and 1,4 billion features.

case	features	F1 score	FPR∗
1	157	0.969 ± 0.01	0.014 ± 0.002
2	1879	0.976 ± 0.009	0.012 ± 0.002
3	2428	0.989 ± 0.006	0.006 ± 0.001

∗FPR: False Positive Rate.



(a) Group 1 F1-measure (b) Group 2 F1-measure (c) Group 3 F1-measure

(d) Group 4 F1-measure (e) Group 5 F1-measure (f) Group 6 F1-measure

(g) Group 7 F1-measure (h) Group 8 F1-measure (i) Group 9 F1-measure

Figure 7: F1-measure obtained by the framework for the 9 input datasets. In red, the classification algorithm is automatically selected by the framework.

Table 7: Comparison with similar solutions. ML reports the number of tested machine learning algorithms; FSR means further security research feature. PRM refers to *permission based analysis*; INT refers to *application intent*; STR refers to *string-based analysis* (Section 2).

API based framework	additional features	ML	FSR	dataset size	mal. apps	F1	FPR
DroidApiMiner [8]	—	4	no	19978	19.9%	97.8%	2.2%
OpSec [9]	PRM	n/a	no	1758	79.5%	97.5%	2.2%
DMDAM [10]	PRM	15	no	170	41.2%	77.0%	n/a
IFE [11]	—	1	no	1100	50.0%	98.1%	8%
Anastasia [12]	PRM,INT	9	no	29864	62.5%	97.3%	5%
SBMD [13]	STR,PMR	6	no	10000	50.0%	97.0%	10%
SeraphimDroid [14]	PRM	5	no	400	50.0%	87.1%	n/a
SafeDroid v2.0	—	6	yes	25346	40.0%	98.9%	0.6%

number that the other solutions employed, the difference in the false positive rate is noticeable. Moreover, our method produced the highest accuracy in terms of F1 score, reaching up to 98.9%. The comparison table also shows a lack of out-of-the-box, customisable solutions in the literature: our framework is the only one available to be employed for further research activities.

5. Conclusion and Future Work

In this paper we presented SafeDroid v2.0, a complete framework for Android apps, to discriminate benign applications from malicious ones. Unlike prior research and other offered solution in this field, our work introduces an open-source, easy to deploy, research framework. Our primary intention was to offer mobile security researchers a technical solution able to conduct the analysis, the formation of the prediction models, and the classification of Android applications. The core of the framework is constituted by machine learning classifiers that are the base of real-time malware detection service for Android devices. Our solution introduces a complete system for static analysis of Android applications, data extraction and storage, comparison of different machine learning classifiers and algorithms, as well as visualisation of the produced results. We also presented a set of experiments showing that the framework is accurate, fast, scalable, and easy to use.

As future work, we aim at implementing a fully automated framework to make SafeDroid v2.0 self-updating, adding a new component that will continuously look up for new malware applications to be analysed and triggers the generation of a new and updated version of the classifier. Moreover, we are planning to perform additional experiments employing different families of malware and different versions of the same malware, in order to prove that SafeDroid v2.0 can also be employed to study the evolution of the different malwares over time. Finally, we are also considering adding another component to complement the static analysis with a dynamic analysis of the apps under analysis.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] IDC International Data Corporation, "Smartphone OS Market Share 2017-Q1," http://www.idc.com/promo/smartphone-market-share/os, 2017.

[2] B. Mike, Android's Piracy Problem Is Forcing Developers To Give Away Games: 'Alto's Adventure' Latest Freebie, 2012.

[3] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[4] J. Oberheide and C. Miller, "Dissecting the android bouncer," in *Proceedings of the SummerCon2012*, New York, NY, USA, 2012.

[5] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[6] R. Goyal, A. Spognardi, N. Dragoni, and M. Argyriou, "SafeDroid: A distributed malware detection service for android," in *Proceedings of the 9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA'16*, pp. 59–66, Macau, China, November 2016.

[7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: automated security certification of android applications," CS-TR-4991, UM Computer Science Department, 2009.

[8] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in android," in *Security and Privacy in Communication Networks*, vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 86–103, Springer, 2013.

[9] A. Ali-Gombe, I. Ahmed, G. G. Richard, and V. Roussev, "OpSeq: android malware fingerprinting," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, ser.PPREW-5, pp. 7:1–7:12, ACM, New York, NY, USA, December 2015.

[10] A. Bhattacharya and R. T. Goswami, "DMDAM: data mining based detection of android malware," in *Proceedings of the First International Conference on Intelligent Computing and Communication*, J. K. Mandal, S. C. Satapathy, M. K. Sanyal, and V. Bhateja, Eds., pp. 187–194, Springer, Singapore, 2017.

[11] M. Yang and Q. Wen, "Detecting android malware with intensive feature engineering," in *Proceedings of the 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 157–161, Aug 2016.

[12] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: android malware detection using static analysis of applications," in *Proceedings of the 8th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2016*, pp. 1–5, Larnaca, Cyprus, November 2016.

[13] A. Martín, H. D. Menéndez, and D. Camacho, "String-based malware detection for android environments," in *Intelligent Distributed Computing X*, vol. 678 of *Studies in Computational Intelligence*, pp. 99–108, Springer International Publishing, Cham, Switzerland, 2017.

[14] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided Android malware classification," *Computers and Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[15] Xiangyu-Ju, "Android malware detection through permission and package," in *Proceedings of the 2014 International Conference on Wavelet Analysis and Pattern Recognition, ICWAPR 2014*, pp. 61–65, Lanzhou, China, July 2014.

[16] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang, "DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features," *IEEE Access*, vol. 6, pp. 31798–31807, 2018.

[17] J.-C. Kuester and A. Bauer, "Monitoring real Android malware," in *Runtime Verification*, vol. 9333 of *Lecture Notes in Computer Science*, pp. 136–152, Springer International Publishing, Cham, Switzerland, 2015.

[18] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, pp. 15–26, ACM, New York, NY, USA, October 2011.

[19] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 5:1–5:29, 2014.

[20] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Computers & Security*, vol. 65, pp. 230–246, 2017.

[21] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proceedings of the Network and Distributed System Security Symposium*, pp. 23–26, San Diego, Calif, USA, 2014.

[22] J. Yu, Q. Huang, and C. Yian, "DroidScreening: a practical framework for real-world Android malware analysis," *Security and Communication Networks*, vol. 9, no. 11, pp. 1435–1449, 2016.

[23] C. Bae and S. Shin, "A collaborative approach on host and network level android malware detection," *Security and Communication Networks*, vol. 9, no. 18, pp. 5639–5650, 2016.

[24] T. M. Mitchell, *Machine Learning*, McGraw-Hill, Inc, New York, NY, USA, 1st edition, 1997.

[25] P. Srikanth, A. Singh, D. Kumar, A. Nagrare, and V. Angoth, "A comparison of machine learning classifiers," in *Advanced Materials and Information Technology Processing*, vol. 271-273, pp. 149–153, Trans Tech Publications, 2011.

[26] H. Kang, J.-W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, Article ID 479174, 2015.