SafeDroid: A Distributed Malware Detection Service for Android

Goyal, Rohit; Spognardi, Angelo; Dragoni, Nicola; Argyriou, Marios

Published in:
Proceedings of the 9th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2016)

Link to article, DOI:
10.1109/SOCA.2016.14

Publication date:
2016

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Goyal, R., Spognardi, A., Dragoni, N., & Argyriou, M. (2016). SafeDroid: A Distributed Malware Detection Service for Android. In *Proceedings of the 9th IEEE International Conference on Service Oriented Computing and Applications (SOCA 2016)* (pp. 59-66). IEEE. https://doi.org/10.1109/SOCA.2016.14

SafeDroid: A Distributed Malware Detection Service for Android

Rohit Goyal*, Angelo Spognardi*, Nicola Dragoni† and Marios Argyriou*
*DTU Compute, Technical University of Denmark. Email: angsp@dtu.dk
†DTU Compute, Technical University of Denmark and AASS Centre, Örebro University, Sweden. Email: ndra@dtu.dk

***Abstract*—Android platform has become a primary target for malware. In this paper we present SafeDroid, an open source distributed service to detect malicious apps on Android by combining static analysis and machine learning techniques. It is composed by three micro-services, working together, combining static analysis and machine learning techniques. SafeDroid has been designed as a user friendly service, providing detailed feedback in case of malware detection. The detection service is optimized to be lightweight and easily updated. The feature set on which the micro-service of detection relies on on has been selected and optimized in order to focus only on the most distinguishing characteristics of the Android apps. We present a prototype to show the effectiveness of the detection mechanism service and the feasibility of the approach.**

I. Introduction

According to techcrunch.com[1], there were 2.6 Billion mobile devices worldwide in 2015 which is expected to rise to 6.1B by 2020. There are applications available for all variety of tasks like entertainment, fitness, health-care, utility, news, magazines, books, movie, music, security, productivity, education, business, finance etc., with millions of apps available in different app stores. Some of these app stores are more strict than others in filtering the apps based on the security assessment. Users, who are mostly unaware of the security implications, tend to ignore this fact and download apps from any source available. This has led to a proliferation of malicious apps in the form of viruses, trojan horses, worms, adware, spyware etc., known for stealing users' data, leaking personal information, running malicious code and degrading the overall performance of the device. Such apps sneak through the app stores due to missing thorough security analysis or they are distributed directly from the servers.

In January 2016, Google's official market place Play Store banished 13 apps as they were making unauthorized downloads and tried to gain root privileges[2]. However, several examples of late discover of malicious apps on the Google Play Store (like Android.Dropdialer[3]) have shown that Google's automated security service can have several troubles on early detection. This is why researchers have invested efforts in defining models and systems to detect malicious applications leaking critical data or carrying out unintended functions.

Malware detection for Android has been conducted with many different approaches, starting from signature based detection, to artificial intelligence, leveraging static, dynamic and hybrid analysis [1]. As it will be further clear, all the approaches have several advantages and weaknesses, but considering the trade-off between usability and effectiveness, static analysis (namely the analysis of the executable code of an app) is still considered to be the most effective approach, mainly when it's bounded with the new paradigm of machine learning (ML) [2], [3], [4], [5]. The most critical step to obtain an effective and efficient ML-based system is the choice of the feature set on which the algorithms have to deal with [6]. In particular, features requiring great overhead to be extracted would lead to poorly usable systems, while ineffective or redundant features would lead to systems with unsatisfying detection rates.

A. Contribution and Outline of the Paper

In this paper we present SafeDroid, an open source distributed malware detection service for Android platforms which is lightweight and suitable for deployment with high speed performances and optimal detection rates. Performance is an important characteristic because it has a direct impact on the user experience. Similarly, detection rate determines the usefulness of the system. Users won't use a service either if the detection takes a long time or if the detection performances are bad (too many false positives or negatives).

SafeDroid is implemented as a set of micro-services working together, combining static analysis and machine learning techniques. SafeDroid inspects the device for installed apps and provides a classification into benign or malicious apps. In case of malicious app, SafeDroid displays the list of feature categories that contribute the most for such prediction. The back end system has been designed as a micro-service which provides classification, based on the API calls of the examined application, and reporting. In particular, a list of 743 API features that describe malicious behavior has been identified, which is a comparatively very small feature set. This set is used for training the machine learning classifier over a data set sourced from prior research.

The novelty of SafeDroid architecture is that the classification is done by a remote micro-service using only the DEX file retrieved from the APK file: the client app saves bandwidth since it only sends the important data, strictly necessary to perform the app evaluation and, thus, significantly reducing the amount of data to be sent. Furthermore, compared to other similar solutions, SafeDroid uses a relatively lower dimensional feature vector to feed the machine learning classifier,

[1]Ingrid Lunden on *techrunch.com*: *6.1B Smartphone Users Globally By 2020, Overtaking Basic Fixed Phone Subscriptions.* http://tcrn.ch/1IbiZNr

[2]Dan Goodin on *arstechnica.com*: *Malicious apps in Google Play made unauthorized downloads, sought root.* http://goo.gl/2jY18e

[3]Steven Musil on *cnet.com*: *Malware went undiscovered for weeks on google play.* http://goo.gl/1I2svr/

while still providing with high detection rates and very few mis-classifications.

In the paper we describe the implementation of the detection service and the evaluation of its performance against three different machine learning classifiers over a data set of real application. We conclude that the Random Forest classifier performs the best with an accuracy of 99.51% and a false positive rate of 0.017.

Outline of the Paper The paper is organized as follows: Section II introduces and analyses the most relevant literature, identifying related weaknesses (which constitute the main motivation for designing a new malware detection micro-service). Section III illustrates the main components of SafeDroid, while Section IV provides further details by motivating design choices. The results of the evaluation experiments of the system are reported in Section V, while Section VI summarizes the paper and provides some further research directions.

II. Related Work

We can group the available literature on malware detection considering three main different approaches, according to the different approaches followed in order for the detection to be performed, namely *static*, *dynamic*, or *hybrid*. With static analysis the applications are evaluated by analyzing their executable code, while with dynamic analysis their classification is performed considering their behaviour, effectively by running them. The hybrid approach, clearly, tries to combine the two analysis methodologies to produce more accurate and precise evaluations.

A. Detection Based on Static Features

Shabtai et al. [7] classifies apps according to a category of utility using the static features extracted from the APK file (the whole package of an application, composed by images, metadata and executable code). The idea is to compare the different feature selection and machine learning algorithms to solve this classification problem and extend it to the domain of malware detection. This early work gave a first insight into the feasibility of this approach to solve security problems.

RiskRanker [8] focuses on detecting zero day Android malware by signature recognition techniques. The authors collected around 105,000 apps from different app stores over a period of 2 months. The prototype is able to identify 718 malicious apps in 29 malware families. This included 322 zero-day malwares. The approach has several limitations, like the reliance on known signatures for the first order analysis, assumption that the attackers use Android libraries for encryption, decryption tasks instead of implementing their own.

L. Batyuk et al. [3] propose a service that statically analyses applications by identifying third party libraries and using rich pattern matching for security warnings. They provide mitigation measures by refactoring binary application packages based on user preferences. The authors present an experimental prototype of the system, but argue that the deployment is only possible by using a third-party hosting server and, then, uploading the APKs to the analysis server for assessment and mitigation, making the approach not very practical.

H. Kang et al. [9] complements the existing static analysis techniques by considering the developer certificate serial numbers of the applications. The authors argue that malware detection can be more effective if the serial number of the app certificate is compared with a predefined blacklist of malicious certificate serial numbers. A data set comprising of about 51,000 benign and 4500 malicious applications has been used for the experiments with a noticeable detection rate of 98%.

J. Saxe et al. [5] base the malware detection on deep neural networks using two dimensional binary program features. The network is directly trained on binary files without any filtering or unpacking. They were able to achieve a 95% detection rate at 0.1% false positive rate.

L. Apvrille and A. Apvrille [2] use a combination of different classification algorithms like SVM, Hidden Markov Models, Logistic Regression etc. The classification module automatically combines various algorithms to produce the best results. The authors tracked 289 features from the static analysis of the APK file which included file properties, Dalvik code properties, resource properties and third party kit properties. The main novelty lies in the classification engine (*Alligator*), which is a free and open-source tool for classifying data.

Y. Aafer et al. [10] proposed DroidAPIMiner, a tool based on the approach that defines a feature vector from the critical API calls, their package level information and their parameters. A data set comprising of about 16K benign and 4K malicious applications is sourced from McAfee and Android Malware Genome project. The authors were able to achieve 99% accuracy and a a false positive rate of 2.2% using k-nearest-neighbor classifier.

D. Arp et al. [11] propose a tool called DREBIN that extracts features by performing broad static analysis on the APK. Properties like suspicious API calls, requested permissions, hardware components, intents, used permissions etc. are taken into account for generating a rich feature set. Linear Support Vector Machine (SVM) is used as learning and classification algorithm. A data set comprising of around 120,000 applications has been used for training and detection. The authors record a detection rate of 94% and false positive rate of 1% for this analysis.

To conclude the overview of the static analysis approaches, we can group the features considered to perform malware detection in the following four categories.

a) Requested permissions: This category considers the list of the permissions requested by an app to access security critical services at the installation time. This is one of the Android security mechanism, since the user is informed and explicitly needs to grant access to these services before they could be used by the app. For example, READ_SMS, SEND_SMS, READ_CONTACTS, WRITE_CONTACTS etc. It has been found that the malicious apps tend to ask for certain set of permissions more frequently than the benign apps [12]. An app requesting access to location information, network communication and personal information is more likely to be sending user location to remote server and thus misusing it.

b) Hardware components: This category, similar to the permissions, considers which hardware components the apps need provide its functioning. A combination of certain features could indicate malicious behavior [10]. For example, an app accessing camera and network connection could send photos to the remote server without user's consent.

c) API calls: This category considers API calls as features requested by an app. For instance, an app can make calls to specific APIs to access services, for example, getDeviceId(), getSubscriberId(), sendTextMessage(). Some of these APIs are more critical than others because of the security and privacy implications of the services they access. The calls to the critical APIs are administered by specific permissions but apps could call APIs without the required permission. This could indicate a malicious behavior suggesting that the app might have tried to use root exploits to call a specific API. Also, a set of critical API calls could define a malicious behavior [10], [11]. For example, an app calling sendTextMessage() API could send SMS to some premium numbers without user's consent.

d) App components: This category considers the four following components of an app: broadcast receivers, content providers, activities and services. These components are the interfaces that the system uses to interact with the app. Certain malware families could use the same component names for malicious activities [11]. This information may help to detect such apps.

B. Detection Based on Dynamic Analysis

DroidRanger (Y. Zhou et al. [12]) uses permission based on filtering and heuristic detection methods to identify malicious apps in the official and non official Android market places. The idea is to filter the apps based on the permissions declared in the manifest file. Only suspicious permissions were used for filtering, such as sending or receiving of SMS. After, the filtered apps are matched with the malware footprints generated from behavioral analysis. It is important to note that behavioral footprints are generated manually by analyzing malware features. The heuristics based detection focuses on identifying apps that exhibit suspicious behavior by dynamically loading code. Dynamic execution monitor is used to inspect the code triggered APIs calls. The system detected 211 malicious apps in the chosen data set, 32 of them in the official market place.

W. Enck et al. [13] proposes TaintDroid, an information flow tracking system for real time user privacy monitoring on smart phones. The system is designed as an extension to the Android OS that tracks the flow of private information through third party apps. The traffic generated from all the installed apps is monitored and scanned for possible data leakage. The goal is to detect if user sensitive data leaves the system through any of these apps. The system makes use of language based security and privacy principles like labeling program variables and files for the purpose of identifying possible leakage of sensitive data. Information regarding transmitted data, application and destination are logged for further analysis. TaintDroid is tested on only 30 apps but is able to provide good results with no false positives.

C. Detection Based on Hybrid Analysis

T. Blasing et al. [14] take into consideration both static and dynamic analysis of applications. The static analysis is based on using pattern matching techniques for known malware behaviour. Dynamic analysis module aims on intercepting system calls to understand the behavior of applications. A Loadable Kernel Module (LKM) has been installed in the kernel space to log system calls, while the application underwent random operations triggered by the end user. The authors tested the system with 150 applications downloaded from the app store and a self written Android malware app.

Y. Zhauniarovich et al. [15] found out that recent Android malware is able to evade the best static analysis tools, due to popular dynamic code update techniques and reflection presence. The authors proposed the use of dynamic analysis techniques along with the existing static analysis to reveal the hidden or updated behavior of the applications. A system called STADYNA has been implemented capturing the advantages of both analyses. The evaluation data set was quite small for the experiment and dynamic analysis steps could not be automated and, thus, have been conducted by manual trigger.

Lindorfer et al. [4] proposed MARVIN to provide a comprehensible and practical malware analysis system to empower the existing technologies, which are dealing with problems like obfuscation and dynamic code loading. The system is based on a hybrid approach, consisting of static and dynamic analysis, leveraging machine learning to provide scores for unknown android apps. The data set consisted of 135,000 Android apps, out of which 15,000 samples were malware. MARVIN reported an accuracy score of 98.24% with less than 0.04% false positives.

D. Limitations of Existing Approaches

Some of the relevant works do not have sufficient data to train and test the model [13], [1]. Other works [10] only discuss the static analysis approaches and the accuracy of the created model, but fail to deploy the system in real world settings. This is particularly important because deployment and integration is a major task in mobile solutions. The apps can be installed from a variety of market places and third party servers. It is therefore important to look at the architecture for real deployment. We overcome these limitations by designing a system feasible for deployment and providing a high prediction accuracy on a data set of 24000 applications.

Usage of machine learning classifiers with static analysis can produce high dimensional feature sets (e.g.: with thousands of features) , as in [11], [4]. A high dimensional feature set usually generates more complex models, which typically suffer from over-fitting. SafeDroid is based on a model trained on a much smaller feature set.

Some of the approaches for static analysis use feature sets consisting of different types of features extracted, like requested permissions, used permissions, hardware components, intents, API calls etc. [11], [2], [4]. Since some of these features have a strong correlation under different categories, these methods miss the interrelationship analysis of the feature selection step. For example, a malicious app sending premium SMS would call API sendTextMessage(), request permission SEND_SMS and make use of it. These three features in the feature set are strongly bind and should be considered only once for training the model. Our solution overcomes this limitation by taking into account only the most relevant API calls.

Most of related work do not take into account dynamic code loading, native code execution and use of reflection, namely those techniques that allow an app to request and access APIs that are not included in the executable code, but provided only at runtime by other applications. In [10], [11], API calls related to this functionality are taken into consideration without any method parameters or the loaded code. In these cases, static analysis performed at byte-code level would not provide concrete information about the behavior of the app. However, dynamic analysis can introduce high latency and also requires real use of the application. Moreover, it is a reactive detection, a method which needs to observe the malicious behaviour on time. We propose SafeDroid as a mechanism to detect malicious apps before their execution and without the need of introducing further latency.

III. SafeDroid architecture

In this Section we introduce the architecture and some implementation details of SafeDroid. The goal is to develop a service suitable for deployment in a real world scenario, able to obtain a good balance between performance and accuracy.

The micro-services on which SafeDroid is built upon, use static analysis in order to limit the classification time, which can be high when using dynamic analysis. Moreover, they need to access the executable code used by an application after its installation. While it would be possible to obtain the APK file of a given application directly from the servers of a marketplace, we do not consider this as a valid option. Accessing the APK directly from an app marketplace, in fact, does not assure accessing the same executable code provided to a given device: we can easily imagine a smart malware distributor able to distinguish between a vulnerable device and a honeypot server and, thus, able to provide different versions of the same application to different clients. SafeDroid has been designed to access the application binary code directly from the device after its installation. Moreover, instead of analyzing the whole APK archive of the app, it only considers the executable part of it, namely the DEX file. This operation allows to save network bandwidth and greatly reduce the communication latency: for example, two known applications (not disclosed) with APK files of 92KB and 1.5MB have the relative DEX files of only 37KB and 810KB, respectively. Finally, since the malicious applications constantly evolve, SafeDroid has been designed in a modular way, so that the detection micro-service can be updated without impacting on the user experience.

With these issues in mind, SafeDroid has been designed and implemented as a distributed service with three key components (Fig. 1): an Android app, a Classification and Reporting Service (CRS), and a Feature Extraction Service (FES).

The first component is a malware detection app for the Android platform, responsible for the interaction with the user. The app retrieves the list of installed applications of the device and allows the user to choose which application to scan. In the Android OS, indeed, the APK archive of every installed app can be accessed with a standard permission, without the device being *rooted*. By means of the "Runtime" Java class,

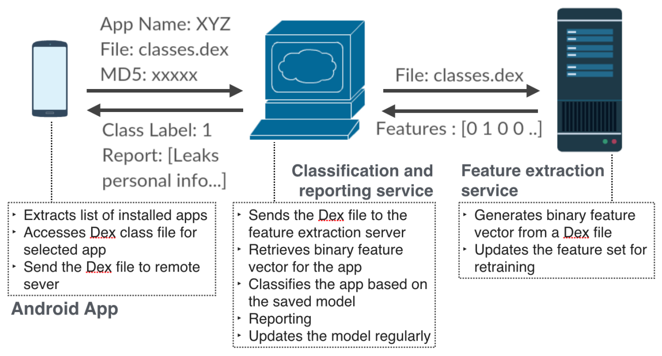

App Name: XYZ
File: classes.dex
MD5: xxxxx
Class Label: 1
Report: [Leaks personal info...]
File: classes.dex
Features : [0 1 0 0 ..]
Classification and reporting service
Feature extraction service
Extracts list of installed apps
Accesses Dex class file for selected app
Send the Dex file to remote sever
Android App
Sends the Dex file to the feature extraction server
Retrieves binary feature vector for the app
Classifies the app based on the saved model
Reporting
Updates the model regularly
Generates binary feature vector from a Dex file
Updates the feature set for retraining

Fig. 1. SafeDroid Architecture: Component Services and Communication

the detector app can extract the DEX file from the APK of the chosen application. The DEX file, which stands for Dalvik Executable, contains the executable code of application. Our app, then, sends the DEX file together with the app name and the MD5 hash of the APK, to the second SafeDroid component, the CRS micro-service. Once the CRS sends back the result of the analysis, in case of malware, the app also includes the reason why the app has been classified as malware.

The CRS micro-service is responsible for the analysis, the classification and interacting with the client app. The CRS is written in Java (based on the Spark Web framework) and acts as an API end-point for the classification and reporting functions. Once it receives the DEX file, it requests the Feature Extraction Service (FES, the third component of SafeDroid, described in the following section) to analyse the file and to return a binary feature vector. This feature vector is then evaluated to assign the class label, namely malicious or benign. The CRS module contains a machine learning classifier (Section IV), based on the Random Forest algorithm implementation in Weka framework [16]. The design of SafeDroid as a distributed service allows to detach the evaluation of the feature vector from the client app in order to easily manage new feature sets. In this way, when we want to adopt a new set of features, we can generate and adopt a new classifier within the CRS service, trained and updated with the current state of malware, without requiring the client app to be up-to-date. The feature set and classifier can also be regularly upgraded, without affecting the user experience, being confined within the CRS service.

As discussed in Section IV, the classifier outputs which features contributed the most to output the malicious label, since it contains a list of ranked features generated by Attribute evaluation method. In particular, features are categorized under several labels depending upon the security and privacy concern associated with them. We have defined six feature categories: suspicious network activity, suspicious ads related activity, retrieves personal and/or device information, suspicious utility method calls and miscellaneous method calls. In order to report the findings, the top 20 classified API calls are retrieved and their most common features are sent back to the end user.

The FES micro-service (implemented in Python) gets the DEX file from CRS and statically analyzes it using a modified version of the open source tool called *Androwarn* [17], as detailed in Section IV. Androwarn is an open source tool written in Python for reverse engineering Android APK files and statically analyzing Dalvik bytecode. It provides functionality for searching packages, method names, method signatures, opcodes etc. in the DEX files. With the outcome of the static analysis, the FES module produces a binary feature vector depicting the APIs present in —and eventually accessed by— the DEX file, which is then forwarded to the CRS micro-service.

IV. SafeDroid Malware Detection Methodology

In this Section we provide further details regarding the detection mechanisms implemented in SafeDroid. As described in Section III, the CRS micro-service relies on machine learning algorithms to classify the applications. To build the classifiers, we adopted an approach similar to [9], in which the classification model is trained with malware samples collected from common malware repository websites such as VirusShare, Contagio Mobile and malware.lu. The employed malware samples were collected during the period of January to August 2013, while the benign apps were downloaded from the official Google Play Store during the same period. It is important to note that these apps are assumed to be benign. We have considered a total of 25000 apps, out of which 4554 (18%) are malware and 20446 (82%) are benign. The number of malware apps is far less than the benign apps, a thing that justifies our data set proportion.

A. *Choice of the Feature Set*

The machine learning model uses data in the form of feature vectors, namely in a structured format able to represent the samples. The examined data will be later transported between the micro-services to determine whether an app is malicious or not. it is also important to provide the correct labels in order to build the model (the training phase of the classifier), alongside the feature vectors. In our case, we wanted to adopt a representative feature set of the malicious activity of the app while, at the same time, having a low number of dimensions, in order not to fall victims of over-fitting and complex models problem. We decided to rely on a binary feature vector, where the occurrence of 0 or 1 determines whether a specific feature is present or not in the app file.

For the detection mechanism of SafeDroid, we opted for features based on the API calls, as introduced in Section II, since they are capable of capturing app behavior very well. For example, API calls like getDeviceId(), getSubscriberId(), sendTextMessage() allow access to the sensitive data or resources of the smartphone: malicious apps try to extract this kind of information without user consent and misuse it. As discussed in [10], [11], the usage of this set of API calls make an app suspicious and thus could be helpful in determining if it should be flagged as malicious or benign. Moreover, other types of features, like requested permissions and hardware components, have a strong correlation to these APIs. Moreover, the API calls can be easily extracted statically from the DEX class file without much computation overhead. This is an important aspect, since the usability of every client solution lies in the fact that it should be fast.

To obtain the desired set of features, an approach inspired by DroidAPIMiner [10] was used. To figure out the relevant APIs that could determine whether an app behaves in a malicious way or not, we can identify the most commonly used API calls from the malware apps. This can be done by searching for all the API calls in the dissembled code (DEX) file of an app. However, this approach provides a very large number of API calls, since in general an app is composed of several dedicated and third party packages and, thus, accesses to a large number of API calls in each of the packages. Also, commonly categorized malware APIs calls are not necessarily malicious in nature. Moreover, the computation to collect the API list for each app and to count the frequency of each API call can be very huge. Thus, our approach makes the computation less expensive and the feature set more relevant, reducing the API call to consider.

Instead of considering all the API calls, we focused mainly on the API calls that are used in the malicious apps more frequently, narrowing down the range of analysis. To filter those critical packages, we used our data set to categorise the packages that are used by malicious apps more often than the ones used by benign apps. For each app, the difference in the count of usage of packages in malicious apps with the benign apps was calculated. The packages which have a percentage difference more than a specified threshold (35%) were considered as the most suitable to distinguish between malicious and benign apps. In particular, we were able to cluster the malicious app in four main categories, namely (1) Telephony and SMS services, (2) Dynamic code loader services (generally used to evade static analysis checks and download and execute code after installation), (3) Network resources services (potentially used to manipulate the responses from the network connections) and (4) System services (used to collect information about the system and the running platform). The total number of packages in our data set counted 2700 packages for the malicious and 3360 for the benign apps. From this, we were able to focus only on 108 packages, ending up having a the set of 743 API calls, constituting our feature vector.

Finally, for each app in our data set, we extracted the binary feature vector and appended a bit representing the class label of the app in its feature vector (malicious or benign).

B. *Feature Set Refinement*

Feature selection (dimensionality reduction) is a step to improve the accuracy of the estimator and the performance on high dimensional data set. A subset of the original features has several benefits, since it can reduce the redundancy of the model. Firstly, it can reduce *over-fitting*: since the dimensionality of the data is reduced, there is less noise and hence less opportunities to make decisions based on that. Reducing the number of features produces a less complex model and consequently more general, also because the duplicate features or features which do not affect classification are removed. However, the feature refinement has to be done considering that it affects *accuracy* while trying to achieve the best trade-off between the model accuracy and its complexity. The assured benefit is the *reduced evaluation time*, since less parameters lead to faster training and classification time, mainly when the size of the application package is high. It is worth to note that machine learning techniques like Random Forest do not benefit much from feature reduction, compared to others like K-nearest neighbor and Support Vector Machines, since do not suffer over-fitting and can handle well high dimensional data [18].

With this in mind, we have used Correlation Attribute Evaluation (CAE) method for feature selection. This method ranks all the features according to their Pearson's coefficient [16]. In this way, we were able to choose to use the 300 top ranked features from our initial 743 features to evaluate the performance of classifiers (see Section V).

C. Model Training and Evaluation

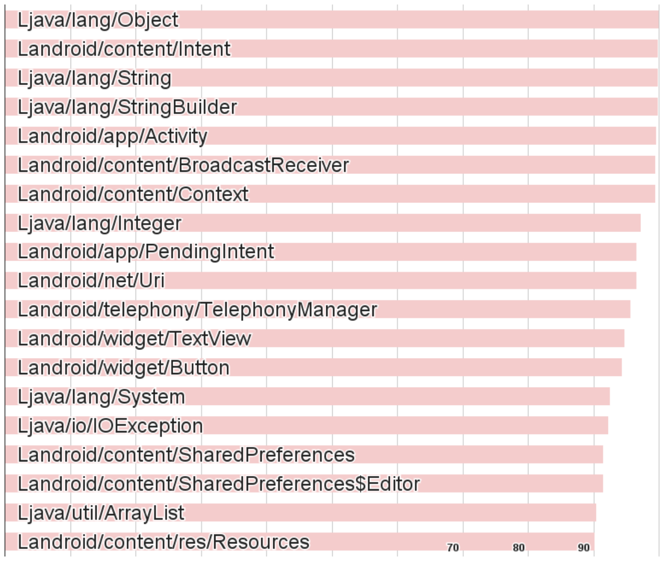
Once the features are extracted, the machine learning classifier can be trained and evaluate its performance. Since SafeDroid prototype has been implemented in Java and we need to use the generated model for classification in the prototype, we ended up using the Weka framework [16]. The feature vector from the last step is fed into the Weka interface and the models for three different classifiers are trained and evaluated. We compared the performance of Random forest, Liner support vector machines and K-nearest neighbor classifiers. The results are averaged over 10 folds of cross validation. The model is, indeed, trained again with the reduced feature set and the results are compared with the complete feature set, as described in the following section.

D. Classifier generation

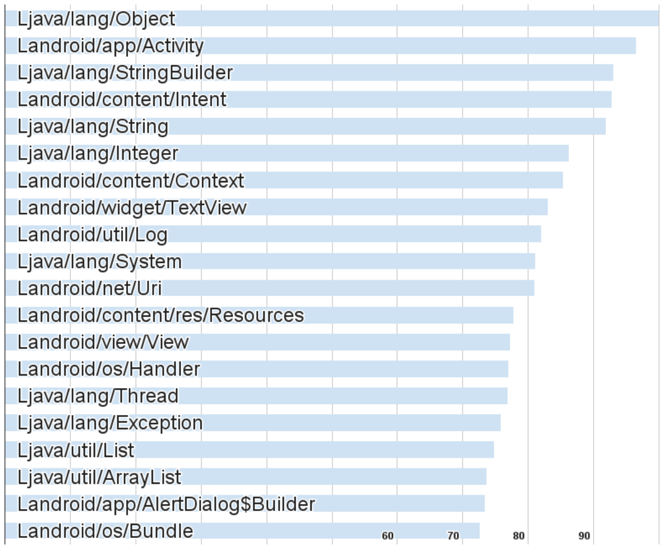
In Fig. 2 we report and compare the top 20 application packages, most used by the Android applications of our data set. In particular, Fig. 2(a) shows the top 20 application packages included by all the malicious apps: the package names are plotted against their percentage count in the malware data set. As we can notice, generic packages like `Ljava/lang/Object`, or like `Ljava/lang/String`, or `Landroid/app/Activity` are used quite a lot by all the apps. This is understandable since these packages are used for usual app flow tasks. Since the usage of such packages does not determine any malicious behavior, we clearly skip the API calls belonging to these packages, as described in Section IV-A. We also notice other packages like `Landroid/telephony/TelephonyManager`, or like `Landroid/content/BroadcastReceiver`, or `Landroid/net/Uri` which have high usage frequency and will be associated to malicious behavior.

Similarly, Fig. 2(b) shows the top 20 packages used by the benign app set. A quick look into the list reveals that the most common packages are `Ljava/lang/Object`, `Landroid/app/Activity`, `Ljava/lang/StringBuilder`; those are the same most used packages of malicious apps in our data set.

Fig. 3 reports on the top 20 packages used in malicious apps more than in benign apps, reporting the difference of the percentage usage count between the packages in malicious and benign apps. The chart is plotted between the package name and the percentage difference of usage in malicious and benign set. Apart from packages related to core functions like `TelephonyManager`, `NotificationManager`, we found many packages related to advertisement services which are more used in malicious apps. As reported in Section IV-A, there is a total of 108 such packages which are used in SafeDroid, consisting in a total of 743 APIs.



(a) Top 20 packages with the highest count in malware data set



(b) Top 20 packages with the highest count in benign data set

Fig. 2. Comparison between packages used by benign and malicious applications of the data set

Exploiting the binary feature vectors generated for all apps, we generated three different classifiers, with three different machine learning mechanisms, namely Random Forest, K-Nearest Neighbor (KNN) and Support Vector Machine (SVM) [18], [6].

V. Evaluation

SafeDroid key characteristics, compared to existing proposals, are feasibility of deployment, performance and accuracy in prediction. This is mainly achieved with the micro-service structure, which helps in the specialization of the single tasks and with their insulation. In this Section, we will evaluate these characteristics over the available parameters.

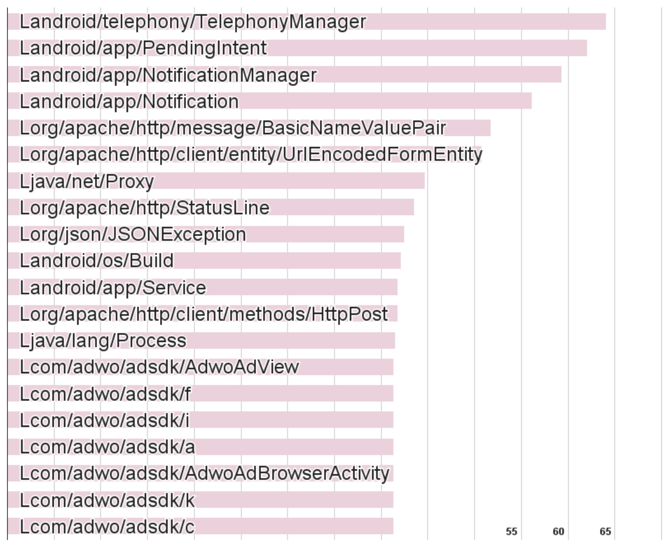


Fig. 3. Top 20 packages with highest difference in usage between malware and benign data set

A. Feasibility

In order to provide evidence of the feasibility of our approach, a proof-of-concept SafeDroid app has been developed with SDK v.23 on Android OS 5.1.1. This Android app is based on the fact that the app can run shell commands on the device using an instance of the "Runtime" class and access the relevant file resource to be sent to the corresponding remote micro-service. To evaluate the performance of the system, we logged the time taken by the API to respond to a client's request, when all both the FES and CRS were running on the same machine. On our testing platform, a machine with 8 GB RAM, 128 SSD hard disk, running Spark web framework on Java 8 on Mac OS X Yosemite, the API responds in 3605 ms. We believe this represents an acceptable response time. Not considering the uploading time of the DEX file, in our experiments we evaluated that around 70% of the execution time is due to the feature extraction phase, 20% to the classification phase and the remaining time to the communication overhead between the SafeDroid micro-services.

We remark that the response time depends on many factors like the size of the DEX file, the number of features, the network bandwidth etc. Being a proof-of-concept, we acknowledge a large margin of improvement, mainly because SafeDroid can highly benefit from its the micro-services based architecture. Moreover, being our main focus the detection rate and the classification performance, we do not explore any computational optimizations, like the fine tuning of the packages taken into account by the androwarn component of the FES service or the use of optimized implementation of the classifiers in the CRS service. However, we are planning to further refine our proof-of-concept and to release a stable and optimized prototype of SafeDroid. The source code of our proof-of-concept distributed service can be found on github[4].

[4]https://github.com/Dubniak/SafeDroid

Classification Algorithm	Accuracy	Precision	Recall	FN rate
Random Forest (trees=10)	99.511%	0.995	0.995	0.005
K-Nearest Neighbor (k=20)	98.74%	0.987	0.987	0.013
SVM (kernel=linear)	97.272%	0.973	0.973	0.027

TABLE I. RESULTS OF CLASSIFICATION USING DIFFERENT CLASSIFIERS WITH 743 FEATURES

Classification Algorithm	Accuracy	Precision	Recall	FN rate
Random Forest (trees=10)	96.95%	0.969	0.97	0.03
K-Nearest Neighbor (k=20)	96.46%	0.965	0.965	0.035
SVM (kernel=linear)	96.04%	0.961	0.96	0.04

TABLE II. RESULTS OF CLASSIFICATION USING DIFFERENT CLASSIFIERS WITH 300 FEATURES

B. Accuracy prediction of classification micro-service

Once the models are obtained with the training data, each of them is evaluated according to *Accuracy*, *Precision*, *Recall* and *False Negative* rates. The results are presented in the Table I, where the statistics are averagely weighted on both the classes of the classification.

The Random Forest classifier with 10 trees gives the highest accuracy of 99.511% and the false negative rate of 0.005, only slightly better than KNN and SVM. One main reason why random forest usually performs better than the other algorithms is that it performs its own feature selection and builds trees with optimized features [18]. False negative rate is a good measure for the model, because classifying the malicious apps as benign can have devastating consequences for the users.

To verify the effectiveness of our optimized and reduced set of features, we compared the obtained results with the classifiers obtained using the 300 top ranked features (ranked by Attribute Evaluation method, as described in Section IV-B) in order to measure the impact of feature selection on the performance of classifiers. The results of the experiment are presented in the Table II.

The Random Forest classifier used in combination with CAE feature selection gives the highest accuracy of 96.95% (Fig. 4) and false positive rate of 0.110 (Fig. 5). Comparing the results (plotted from Table I and Table II), we observe that the accuracy of the classifier dropped by a small 2.56% while the number of features decreased by over 50%, since the feature vector changes from 743 to 300. Considering the false negative rate, we observe an even smaller change; it drops to 0.03 from the initial value of 0.005 for the Random Forest classifier. A similar observation can be done for the other two classifiers. These experiments confirm that we can use the refined set of features while SafeDroid is still able to produce highly accurate classification results of malicious and benign apps.

VI. CONCLUSION AND FUTURE WORK

With the widespread usage of smart-phones and the ease of spreading malware, the Android platform has become a prime target for the attackers. In this paper, we have presented SafeDroid, an open source distributed malware detection service for Android. SafeDroid relies on a simplified analysis of the DEX file for an app which is used for extracting static features. The feature selection has been specifically performed

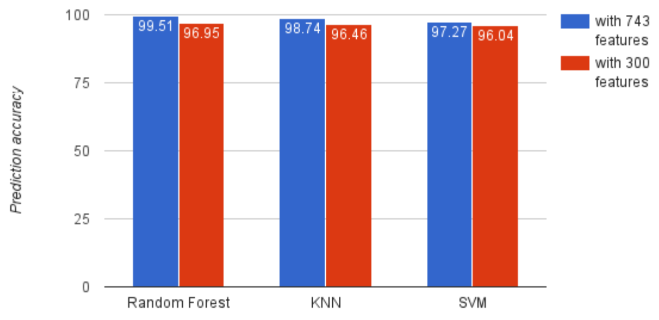


Fig. 4. Accuracy of different classifiers with and without feature selection

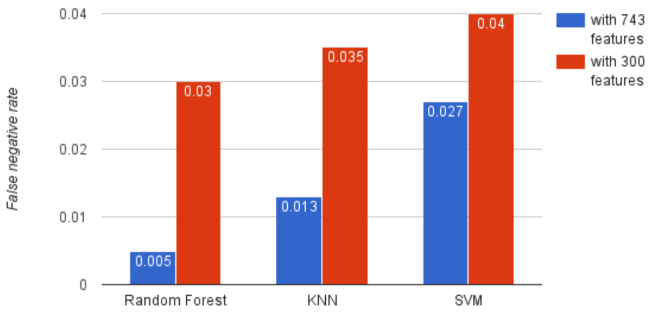


Fig. 5. False positive rate of different classifiers with and without feature selection

in order to have an optimized set of features, composed by the most distinguishing ones. By means of machine learning classifiers, we have built a proof-of-concept prototype able to provide a high accuracy and low false negative rate. The SafeDroid architecture allows to detach the update of the classifiers (running as a separate remote service) from the client app running on the user device, also delivering high performance, to improve the overall usability.

We are currently working to make the system fully automated, in order to continually update its classifier, evaluating the API calls of newly provided applications, in an almost real-time fashion. Moreover, we will continue to extend the data set of malware SafeDroid relies on, considering the continually changing and evolving scenario of malicious software for Android. Finally, we plan to integrate an additional micro-service based on dynamic analysis, to be optionally invoked by the user. This would support the CRS during the detection phase and restrain the risks of reflection and dynamic code loading.

Acknowledgments

We thank the anonymous reviewers for their useful comments and suggestions.

References

[1] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *J.C.V.H.T.*, pp. 1–12, 2015.

[2] L. Apvrille and A. Apvrille, "Identifying unknown android malware with feature extractions and classification techniques," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 182–189.

[3] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications," in *MALWARE'11*. IEEE, 2011.

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

[5] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," *arXiv:1508.03096*, 2015.

[6] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.

[7] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying android applications using machine learning," in *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE, 2010, pp. 329–333.

[8] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[9] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *I.J.D.S.N*, vol. 2015, p. 7, 2015.

[10] 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*. Springer, 2013, pp. 86–103.

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[12] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.

[13] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM TOCS*, vol. 32, no. 2, p. 5, 2014.

[14] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *MALWARE'10*. IEEE, 2010.

[15] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 37–48.

[16] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD*, vol. 11, no. 1, pp. 10–18, 2009.

[17] "Androwarn," https://github.com/maaaaz/androwarn, accessed : 2016-02-04.

[18] A. Cutler, D. R. Cutler, and J. R. Stevens, *Random Forests*. Boston, MA: Springer US, 2012, pp. 157–175. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-9326-7_5