Noname manuscript No.
(will be inserted by the editor)

MOCDroid: Multi-Objective Evolutionary Classifier for Android Malware Detection

Alejandro Martín · Héctor D. Menéndez · David Camacho

Received: date / Accepted: date

Abstract Malware threats are growing, while at the same time, concealment strategies are being used to make them undetectable for current commercial Anti-Virus. Android is one of the target architectures where these problems are specially alarming, due to the wide extension of the platform in different everyday devices. The detection is specially relevant for Android markets in order to ensure that all the software they offer is clean, however, obfuscation has proven to be effective at evading the detection process. In this paper we leverage third-party calls to bypass the effects of these concealment strategies, since they cannot be obfuscated. We combine clustering and multi-objective optimisation to generate a classifier based on specific behaviours defined by 3rd party calls groups. The optimiser ensures that these groups are related to malicious or benign behaviours cleaning any non-discriminative pattern. This tool, named MOCDroid, achieves an accuracy of 94.6% in test with 2.12% of false positives with real apps extracted from the wild, overcoming all commercial Anti-Virus engines from VirusTotal.

Alejandro Martín
Universidad Autónoma of Madrid
Tel.: +34 91 497 7535
Fax: +34 91 497 2233
E-mail: alejandro.martin@uam.es

Héctor D. Menéndez
University College London
Tel: +44 (0)20 3108 4042
Fax: +44 (0)20 7387 1397
E-mail: h.menendez@ucl.ac.uk

David Camacho
Universidad Autónoma of Madrid
Tel.: +34 91 497 2288
Fax: +34 91 497 2233
E-mail: david.camacho@uam.es

Keywords Android · Malware · Clustering · Classification

1 Introduction

The Mobile malware report released by G DATA[1] for the last quarter of 2015 offers a panorama of the evolution and current status of malware affecting smartphones. Android, with a great market share, is a potential target for many virus writers. In 2015, the incredible figure of 2,333,777 new malicious applications was reached, which is an increase of 50% in comparison with the previous year. These numbers make clear the necessity of fast tools capable of detecting zero-day malware. This observation is reinforced by the fact that many Android app stores contain malware in their application range [31].

There are two main approaches when analysing applications and determining their malicious or benign character, namely static analysis and dynamic analysis. While the former is focused on extracting characteristic data from the file containing the application, dynamic analysis concentrates on its execution, monitoring the actions performed. This latter provides a deeper analysis of the application, however, entails serious costs in addition to the need for safety precautions. Static analysis, on the other hand, allows to analyse applications more quickly, but is heavily affected by the use of obfuscation techniques [28], which are able to mask the true purposes until the application execution [19]. Dynamic analysis is also susceptible of being useless when the application is able to detect that it is being analysed [20].

[1] Available at: https://secure.gd/dl-us-mmwr201504

Although Java code is highly vulnerable to obfuscation, this is not possible with third-party calls [7]. Any reference to a Java library needs to remain unchanged in order to allow the execution of the application. The method presented in this paper takes the import terms contained in the Java classes of Android applications to create two behavioural patterns, one for malicious and one for benign applications. These patters, consisting on groups of import terms that use to appear together in the same applications, are compared against the list of imports of a new application, where the most similar pattern will decide its nature. A clustering algorithm [6,17] is in charge of creating these groups, separately for malicious and benign applications. The selection of the groups that are actually representative of particular behaviours is addressed by a multi-objective genetic algorithm, which enables or disables clusters seeking the maximum accuracy rate and the minimum number of false positives rate in a test dataset.

Previous research focused on static analysis on Android have used different characteristics of the application, such as API-Level features [1] or permissions [3], among others [24]. The proposed method uses third-party calls as a novel technique to discern particular benign and malicious behaviours. Due to the large amount of data required for analysing this characteristic, an algorithm able to manage large volumes of data is needed. Genetic algorithms have proven to be effective with complex and large datasets, in a wide variety of domains as it can be malware detection based on static analysis [15]. The method presented in this paper takes advantage from the use of SPEA2 [32], a multi-objective genetic algorithm, to select groups of import terms that allows to discriminate distinctive behaviours of malware and benign-ware.

The rest of this paper is organised as follows: the next section offers a description of the related work, subsequently, a section presents the method proposed, detailing the analysis of the import terms extracted from Android applications and defining the classification model, which is followed by the experimental setup and the description of the experimentation. The final section provides conclusions and possible future research.

2 Related Work

Android malware analysis has been performed from two main perspectives: detection and classification. The former aims to discriminate between malware and benignware [27], while the latter aims to generate a model that can detect specific malware families [2]. In this work we focus the analysis on the detection problem. This section presents some background about malware detection in general and, specifically, in Android devices.

2.1 Malware analysis

Understanding malware has become a complex problem over the last few years. The definition of malicious behaviour has a fuzzy boundary between those programs that can be considered benign programs and those consider malware. Out of this boundary, we normally categorise maliciousness based on known behaviours extracted from a deep software analysis [4]. Once we are able to determine that a specific behaviour is harmful, we use that information to detect similar behaviours in order to understand whether a new program is malicious or not. The main analysis strategies for understanding malware behaviour are [9]:

– **Static Analysis**: these kinds of analyses study a program from its source code or disassembly code. In Android case, the starting point is the dex code. The common static procedures extract the program control flow graph, the opcode (operation codes) and variables variations, among others, and study the behaviour of different branches in order to identify suspicious code.
– **Dynamic Analysis**: these approaches are based on studying the malware according to its execution behaviour. The frequent setup is based on an emulator or virtual machine running the application. Different information is extracted from this process, specially focused on traces, network packets, memory modification and register modification, among others.
– **Hybrid Analysis**: These methodologies combine static and dynamic analysis in order to complement each other.

Due to the development of new malicious behaviours requires a strong effort [9], black hats have developed different ways to overcome the detection process. These concealment strategies are usually applied in different levels of abstraction in order to avoid static and dynamic analysis.

For static analysis, the concealment strategy aims to cheat the disassembly process and the control flow graph generation. This is usually performed using polymorphism and metamorphism. Polymorphism consists on changing the appearance of the program, using, for example, encryption or compression of specific parts [9]. Metamorphism is focused on recoding the program, which is usually performed using obfuscation, specially on the control flow graph [9]. Another concealment strategy is based on dynamic classes which are usually hidden using polymorphism. These classes are generated during runtime. In this paper we are not targeting this last case, but the previous ones.

In dynamic analysis, the concealment is based on hiding the malware behaviour at least long enough to skip the detection. Furthermore, black hats have methods for identifying the environment where the application is running, and their malware uses this information to perform only benign operations. This is usually applied for Android emulators or virtual machines [25]. The combination of static and dynamic concealment strategies can also avoid hybrid analysis.

Even when the concealment is a strong limitation for the analysis, there are some limitations to the concealment, such as system calls, that can not be obfuscated [7] and provide enough information about the behaviour of a program, specially when this program aims to control the system or use it in a harmful way, which is usually the main goal of malware. This work is focused on this specific information to detect Android malware.

2.2 Malware Detection in Android system

The first references to malware designed for Android date back to the same year it was presented the first device running this Operating System, by the year 2008. From then on, the number of malicious applications has grown exponentially and, as a result, important efforts have been performed in order detect them and reduce their effects.

Several authors have dealt with the malware detection problem from different perspectives. For example, Shabtai et al. [23] introduce a framework which extracts execution features and applies Machine Learning techniques to classify them. In a similar way, Arp et al [2] combine different classifiers with execution features creating a new tool, named Drebin. Another similar example is CooperDroid [27] which is focused on reconstructing the behaviour of an Android application in order to identify Malware. CopperDroid is a dynamic analysis tool where system calls are analysed dinamically, which make the application less vulnerable to alterations. Our approach is similar in that sense, but we use static system calls, instead of dynamic, which can be hidden.

Other example is presented by Isohara et al. [10] which focus on Android markets and its security during the apps validation process. This work shows that some markets do not try to detect malicious apps, compromising devices and sensitive data. Authors create an audit framework to monitor the application behaviour recording all the system calls invoked and using signatures of normal calls. The technique used to detect the Malware behaviour is based on kernels. DroidSIFT [29] proposes a similar semantic-based methodology, based on API Dependency Graphs. Authors use a similarity metric between the API Dependency Graphs of different applications and a threshold to detect anomalies. Also their method aims to detect different malware families.

Research has also studied Obfuscated malware, analysing its effects in the code. For example ALTERDROID [26] defines an approach for detecting obfuscated components. A deep study of the implications of using these concealment strategies is presented in [30], showing that specific families of malware (for example the DroidKingFu family) make use of encryption to hide the malicious part of the application. These obfuscation techniques are able to adversely affect the precision of commercial Anti-virus [21]. As we will show in the experimentation section, our method is able to detect more accurately malicious applications in comparison with commercial Anti-virus engines.

Machine Learning has been successfully applied to the Android malware detection problem using different features, such as binary information [13], Control Flow Graphs [22], analysing Dalvik bytecode frequency analysis [11] or API-Level features [1]. In addition, these techniques allow to defeat the use of obfuscation techniques [14]. The selection of features is a very important step when using Machine Learning tools [16], where API calls and Control Flow Graphs are the most discriminative to generate strong classification models. The method presented in this paper makes use of these techniques to build a strong classifier able to reach high accuracy rates even when obfuscation techniques are involved.

From a human-based perception analysis, Gorla et al. developed a detection framework named CHABADA which is able to detect malware using the reviews published by the users in the application market [8]. This framework is more robust in some cases than usual analytical process, but it is extremely sensitive to users perceptions. The tool presented in this work aims to overcome all these limitations.

3 MOCDroid: the Multi-Objective Evolutionary Classifier

The third-party calls invoked from an Android application can be seen as a set of relevant characteristics that may allow to create general patterns to discriminate between malware and benign-ware samples. Due to these calls target external software, it is difficult to obfuscate them and most of the authors avoid it. The combination of different calls help to define a behaviour, e.g., when an app tries to activate the microphone and the network, it is clear that the application is sending audio data to an external server. These behaviours can be statically approximated by the semantic intention extracted from 3rd party API calls combinations, which can be described using clustering over these calls. Once we define these groups, the final model, named MOC-Droid, needs to discriminate malware and benign-ware, hence, it creates two submodels keeping only relevant behaviours for these two categories. A candidate program will be evaluated against these two submodels to measure if it fits better with benign-ware or malware. The whole workflow of the process is summarised in Fig. 1, described in depth in the next sections.

3.1 Mining Android imports from source code

Our main hypothesis, which claims that the imports terms included in each Java class of an Android application can be used to arise different behaviours of malware and benign-ware samples, involves a series of preprocessing techniques to extract the needed information of each app. This will help to create a representation of each malicious or benign application to train a model through a Multi-Objective Optimisation process, with the ultimate aim of generating two representative model of each kind of applications.

As stated before, we have selected the specific declaration of imports terms to represent each application. Thus, the first big step involves analysing each app separately to retrieve all these terms. Android application are presented as files with the ".apk" extension, which are ZIP files that can be **decompressed** with any decompression tool. Each of the applications retrieved was uncompressed with Unzip to extract its DEX files, which contain compiled code to be executed in the Dalvik Virtual Machine. Using Jadx[2], this code is **decompiled** to obtain Java code (similar, but not necessarily identical, to that written by the authors due the use of obfuscation techniques), which allows to finally **retrieve** a list of the import terms declared in the application.

3.2 Clustering imports

This set of import terms forms a set of raw information which can hardly be used to disclose specific behaviours of malware and benign-ware; import terms can be widely dispersed, complicating the search of general patterns able to discern between malicious and benign applications. While a specific class (declared by an import term in the Java source code header) can be used in different applications without providing any relevant information, an association of import terms that tend to appear together (e.g. a function which implies calling another particular function), may arise more general patterns and thus facilitate the creation of a malware and benign representative model.

The process of grouping this terms can be addressed using clustering techniques [12], allowing to find recurring groups of different sizes of correlated import terms. We have to note that, thus far, each sample is represented by a variably sized list of import terms declared. Applying a clustering process over a dataset involves representing all samples in the same space, that is to say, with a shared configuration of attributes (a vector of equals length for all of them). As each application is linked to a particular set of import terms, these sets can be combined in order to generate a new binary vector of attributes listing all the import terms found. The values of this vector will indicate whether the application includes a particular import term ("1") or not ("0"). This process is equivalent to that used for generating a Term-Document matrix, where each document represents an app and a term represents an import term. Given this equivalence, we used the Text Mining Package for R [18] to generate this matrix. This package also includes a function to remove sparse terms, only employed on a few documents (or applications), based on a parameter that allows to adjust the sparsity.

Once these two matrices are built, one formed by malware and one by benign samples, are then passed as input to the clustering algorithm, coupled with the numbers of clusters desired. Throughout the process, an euclidean distance will be used to measure the proximity between samples that, given the representation chosen, considers as closer those samples that are shared by a set of applications. This clustering process is executed two times, one to group sets of imports terms of benign-ware applications and a second one with terms related to malicious ones, which will be the input of the next step, a genetic algorithm.

3.3 Multi-Objective Optimisation

The basic idea of the classification algorithm developed consists on generating two representative models, one for malware and another one for benign-ware. Each of these two models will be related with a certain configuration of clusters, a subset of the clusters delivered in the previous step for the respective model. When a new sample is presented to be classified, its import terms will be compared separately with each representative model and the most similar one will deliver the label of the sample. This classification model has been

[2] https://github.com/skylot/jadx

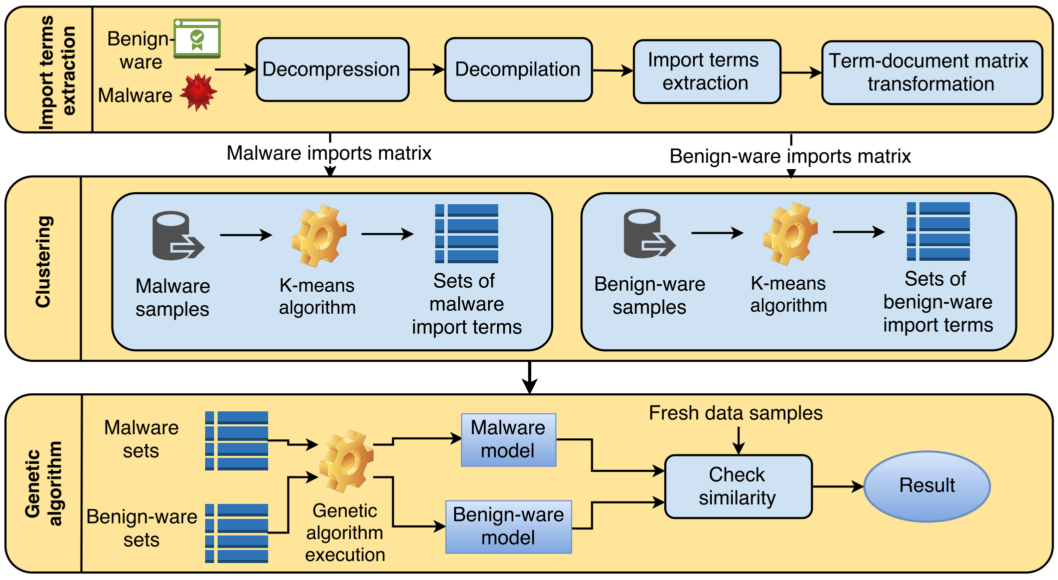


Fig. 1: General diagram of the classification model

implemented following a Multi-Objective Optimisation strategy leaded by a genetic algorithm, where each individual represents a possible solution, simultaneously defining both configurations of clusters (sets of import terms sets) enabled and disabled representative of malicious and benign applications. Below is described the design chosen for this genetic algorithm.

3.3.1 Encoding

When designing a genetic algorithm, one of the most important steps is the individual's encoding, since it will define the shape of the solutions. In the present case, each individual must consider two models at the same time. For this reason, it is divided in two sections called segments, representing respectively the benign and malicious model. These segments contain a number of positions equivalent to the number of clusters used in the associated clustering model, m in the case of the malware clustering process and n in the case of benign-ware. Each position of the segment is related with a particular cluster, where it is only possible to adopt two different values to indicate whether the associated cluster is enabled (its import terms are taken into account to classify samples) or disabled (its import terms are discarded).

3.3.2 Genetic operations

There are four operations that lead the evolution of the population composed by these individuals. A **selection** operator, which is elitist, chooses the l best individuals of a generation to be part of the next one; a Tournament operator is in charge of selecting the individuals to be **reproduced**; a **crossover** operator performs a uniform crossover separately in each segment of the individual and, finally, a Flip Bit based **mutation** is implemented to swap the state of each cluster between enabled or disabled.

3.3.3 Fitness function

Enabling or disabling different clusters will lead to two discriminative models specifically adapted to malicious and benign applications. When analysing a new sample, it will be compared against each model to check their degree of adaptation and the model that best suits the sample will decide the prediction. Comparing the prediction of a example in the training dataset to its real label will allow to assess the quality of each solution. If this process is performed with all the samples in the training dataset, it is possible to measure the accuracy of the classification model: the percentage of instances correctly classified. Although this measure is greatly significant of the quality of the model, when evaluating a malware detection tool it is also advisable to evaluate the false positives rates, responsible of ensuring the correct operation of system applications. These two values, have been chosen to determine the fitness value of an individual, linking each one to a different objective of the optimisation process: while the first one tries to maximise the accuracy, the second one seeks to provide solutions with a low false positives rate.

To explain the prediction procedure in detail, its pseudo-code is shown in Figure 1. This function receives the individual, a candidate program and the two im-

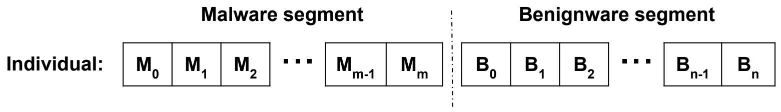

Malware segment
Benignware segment
Individual:
M_0 M_1 M_2 ··· M_{m-1} M_m
B_0 B_1 B_2 ··· B_{n-1} B_n

Fig. 2: Representation of the individual

ports models: malware and benign-ware. A loop iterates over each position of the whole chromosome (line 2). Since it is divided in two segments, in each iteration it is evaluated if the position of the chromosome being analysed belongs to the first segment (line 3), associated to the malware model, or to the second segment (line 8), linked to the benign model. The following steps are similar for both segments: for each cluster, represented by each position in the chromosome, it is checked if it is activated (line 3 or 8 depending on the segment). If yes, it is retrieved the list of imports terms of the cluster in question (line 4 or 9), comparing them to the list of imports presented in the candidate sample (line 5 or 10). If the cluster is represented within the list of imports declared in the application, the related counter is increased by 1. The segment providing a higher number of clusters represented in the sample delivers the prediction.

This prediction function is applied to each sample in the training dataset, and compared to the real label enabling to return the two values that define the fitness value of the individual being evaluated: the accuracy as the percentage of samples correctly labelled and the false positives rate. Throughout the evolutionary process, these values will lead to solutions where the first objective is maximised while the second one minimised. At the end, different solutions forming a Pareto frontier will be returned. Because most of the approaches proposed for Malware detection consider the accuracy as the main factor to assess its quality, we have extracted from the Pareto front the solution maximising the first objective.

Algorithm 1 Candidate Prediction Procedure

```
Require: Chromosome, MalModel, BenModel, Candidate
Ensure: Prediction: Malware or Benign-ware
 1: CountMal = 0, CountBen = 0
 2: for i = 0 to len(Chromosome) do
 3:     if Chromosome[i] == 1 && i <= len(MalModel) then
 4:         imports = getImp(MalModel,i)
 5:         if imports ⊆ candidate then
 6:             CountMal++
 7:         end if
 8:     else if Chromosome[i] == 1 && i > len(MalModel) then
 9:         imports = getImp(BenModel,i−len(MalModel))
10:         if imports ⊆ candidate then
11:             CountBen++
12:         end if
13:     end if
14: end for
15: if CountBen < CountMal then
16:     return "Malware"
17: end if
18: return "Benign-ware"
```

4 Experimental Setup

This section describes the set of Android apps we have collected and processed to train and test the model, as well as the parameterisation of the classification algorithm and the preprocessing operations performed.

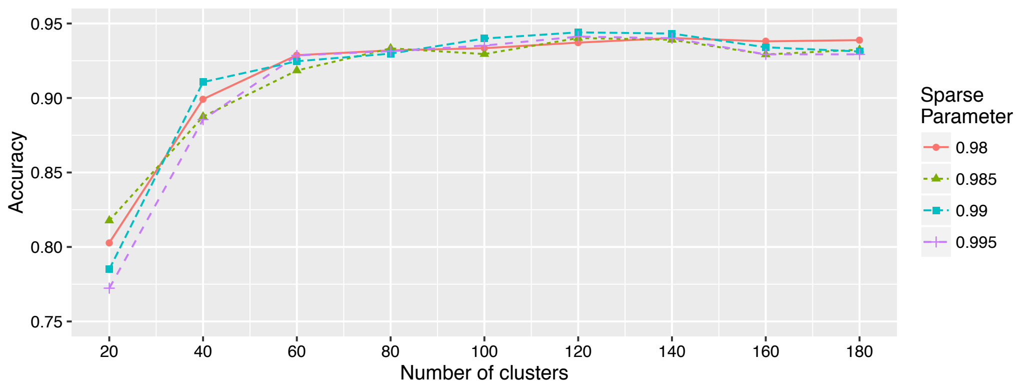
4.1 Dataset

The assessment of the classification model described above requires a set of Android applications labelled as benign and malicious. We have collected 9,430 benign and 9,383 malign Android applications. The first ones have been extracted from an on-line app store called Aptoide[3]. On the other hand, the malign applications have been obtained from VirusShare[4], an on-line portal that provides malicious applications for research purposes. While the applications retrieved from VirusShare can be considered certainly as malware, since they have been analysed via the VirusTotal[5] website, the nature of the apks downloaded from Aptoide is unknown. For this reason, the original obtained from this website was also analysed with VirusTotal, discarding those applications that tested positive for malware (they did not joint the malware set due to the lack of certainty of their categorisation). The data have been divided into two equally-sized partitions with a balanced number of benign and malicious applications, one dedicated to train the algorithm and the other to test it.

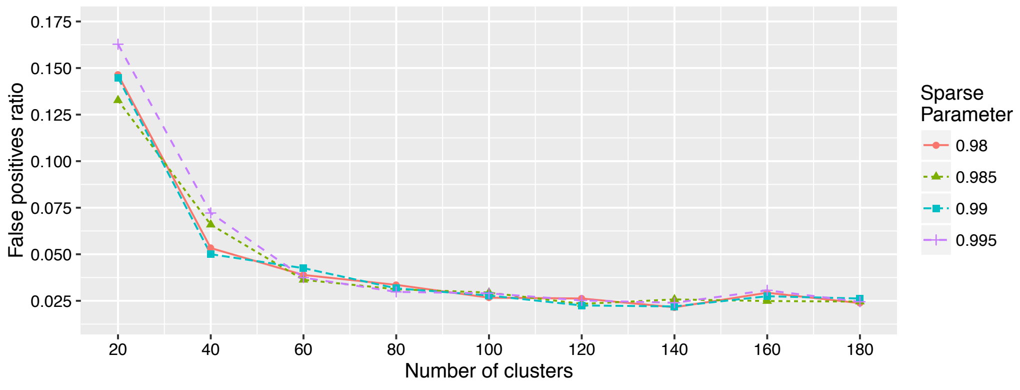
4.2 Genetic Algorithm Parameterisation

The selection of the right parameters for the execution of the genetic algorithm was performed using a Grid

[3] http://www.aptoide.com/
[4] https://www.virusshare.com/
[5] https://www.virustotal.com



(a) Accuracy ratio



(b) False positives ratio

Fig. 3: Accuracy and false positives ratios depending on the number of clusters

Search, testing different configurations in order to find the most appropriate settings. The population size was fixed to 500 individuals with a range of 10 to 1000, while the maximum number of generations was set at 200 after testing that it was enough to ensure the convergence. Regarding the crossover and mutation probabilities, they were fixed 20% and 10% respectively, with a range of 10% to 40%, and the elitism was defined to take the best 10 individuals of each generation with a range between 5 to 50.

Due to the random nature of genetic algorithms, for each different configuration, the genetic algorithm run 50 times. Regarding the possible configurations to test, two parameters were identified as affecting considerably the results: the number of clusters and the number of import terms used. While the first one is a parameter of the clustering algorithm, the number of terms depends on a function of the TM package for R, which is in charge of removing sparse terms. A value ranging from 0 to 1 determines the permissiveness with these sparse terms, where a high value is related to the use of more sparse terms. We have executed different configurations based on the parameter of the remove sparse function ranging between 0,980 and 0,995. Higher values were impossible to test, due to the large amount of time and memory needed. Table 1 shows the final size of the datasets after applying different values to these function. The second parameter having large effects on the results is the number of clusters. We have executed first the k-means algorithm with different numbers between 20 and 180 clusters, pursuing a high level of granularity but without losing generalisation ability.

5 Experimentation

The large number of applications collected and analysed allows to make a detailed assessment of the algorithm proposed. This section describes the evaluation performed with this dataset, explaining the results obtained, providing an evaluation of different parameter configurations with the aim of choosing the most appropriate, describing a comparison against other malware detection tools to, finally, discuss about the quality and relevance of the results.

Sparse	No. Import terms		Best setting		
parameter	Malware dataset	Benign-ware dataset	Accuracy rate	False positives rate	Number of clusters
0.995	5,938	18,283	94.26%	2.35%	120
0.990	3,132	14,003	**94.60%**	**2.12%**	120
0.985	1.873	11,861	93.90%	2.44%	140
0.980	1,484	9,222	93.90%	2.16%	180

Table 1: Malware and benign-ware datasets sizes and results achieved depending on the parameter of remove sparse function

5.1 Parameters configuration and selection

Evaluating a method dependent on different parameter involves testing each possible configuration in order to select the most appropriate with a view to maximise the results. These configurations mainly depend on two parameters: the number of clusters in which the import terms of each label are grouped and the argument of the function in charge of removing sparse terms. Both number have an import influence in the solutions, affecting considerably its complexity. An increment in the number of clusters is related to a more complex classification model, since the individual's encoding becomes larger at the same time that the average cluster size decreases, focusing on specific import terms instead of using general behavioural patterns. In the other side, the sparse parameter is completely correlated with the number of import terms used. Using the lowest possible quantity, taking into account the most important ones, will help to generate classification model more robust against fresh data.

Each possible value assignment to these two parameters, based on the ranges detailed in the previous section, was executed 50 times. Figure 3 shows the average accuracy and false positives rates for these executions depending on the parameter of the sparse function and the number of clusters. This last value plays an essential role in the achievement of the best solutions, both in terms of accuracy and false positives rates. Both measures improve considerably when this value is increased from the lowest value to around 80 clusters. At that point, the quality of the individuals is stabilised and few changes are reported. This suggests that both the malware space and the benign-ware space of samples need at least a division into 80 regions to separate representative groups of calls of a particular nature. At the same time, from 140 clusters onward, neither the accuracy nor the false positives ratio improve. This means that is not necessary to create a further subdivision of the space, which is not advisable if it is pursued a model with a high generalisation ability.

Regarding the number of imports terms used, defined by the argument of the remove sparse function, the results remained largely unchanged for all the values tested. Discuss the relevance of this fact requires first to measure the effects of the parameter chosen in the number of terms. Table 1 the resulting number of import terms both in the malware dataset and in the benign one. Decreasing the parameter from 0.995 to 0.980 results in a reduction of the import sets related to malware by a factor of 4, while in the case of benign-ware, the list of imports is reduced by half. Going forward, Fig. 3 shows that, despite the drastic reduction of import terms, there is hardly any difference in the results, meaning that there is a subset of terms of the whole dataset able to distinguish between samples with high precision.

Table 1 also shows the best results achieved depending on the sparse parameter used for the best number of clusters setting found. Each value is obtained from the average of the 50 executions performed, then, the number of cluster which maximises the accuracy in the training dataset is used to show the considered as best solution for each sparse parameter value. The best result, both in terms of accuracy and false positives rate, is achieved using 0.990 as argument of the remove sparse function and grouping the import terms of both labels in 120 clusters, which arises a 94.60% accuracy with 2.12% of false positives rate.

5.2 Comparison against Anti-virus engines

With the objective of assessing and comparing these results, we have analysed a random set of 10,000 benign and malicious applications (the same size as the used in the test dataset) with the different Anti-Virus engines provided by VirusTotal. Table 2 summarises the best results obtained with these Anti-Virus in comparison to MOCDroid. The best engine value arises a 83% accuracy, which is far from the number obtained with the method here described, more than 10% below. This large gap can be produced by the effects of obfuscation techniques, which these engines are not able to successfully tackle, and also due to the use of non sufficiently discriminatory characteristics. The use of third-party calls allows to successfully neutralise the changes produced by these concealment strategies, focusing on highly representative and inmutable features. Thus, the selection of specific

Malware detection engine	Accuracy
MOCDroid	**94.60%**
Cyren	83.03%
Ikarus	82.72%
VIPRE	82.53%
McAfee	82.45%
AVG	82.36%
AVware	81.95%
ESET NOD32	81.81%
CAT QuickHeal	81.79%
AegisLab	81.74%
NANO Antivirus	81.15%

Table 2: Comparison of results between MOCDroid and the top ten accuracy results for 10 different commercial engines applied by VirusTotal

groups of import terms helps to reveal behaviours associated mainly to a particular type of applications, whether malware or benign-ware.

5.3 Discussion

As the results have stated, using third-party calls by the declaration of import terms in the application allows to arise high accuracy and low false positives rates. In comparison with 10 commercial Anti-virus, the method here presented is improves the results by more than 10 percentage points. The impossibility of obfuscating third-party calls make it possible to avoid the effects of the concealment strategies that hinder the malicious code in order to be detected by a static analysis.

The approach followed in this research allows to make a series of conclusions based on the different results obtained. Recalling the plots shown in Fig. 3, reducing remarkably the number of import terms produces very few changes in the two quality measurements used. This fact can be understood by assuming that the most important differences between benign and malicious applications follow high-level behavioural patterns, instead of focusing in particular details. This is also happening with the number of clusters, which can be fixed to a maximum of 140 groups to arise the best results, which can be considered as the number of groups in which import terms related to malware or benign-ware have to be grouped in order to distinguish relevant differences.

6 Conclusions

Although obfuscation techniques represent a barrier to detect the nature of a program, it is still possible to arise distinctive characteristics which allow to differentiate behaviours between malicious and benign applications. In this research we have developed a method which uses the 3rd party API calls included in each Java file, which cannot be obfuscated, to create an accurate classification model, named MOCDroid, to disclose sets of import terms typically associated to malware and benign-ware. The core of this method, a multi-objective genetic algorithm, is able to successfully select the most representative groups of these two types of applications, pursuing high accuracy rates and minimising the number of false positives. Our future work involves taking into account more characteristics and the use of other clustering methods with the ultimate aim of improving the accuracy and face big data approaches [5].

7 Compliance with Ethical Standards

Author Alejandro Martín declares that he has no conflict of interest. Author Héctor D. Menéndez declares that he has no conflict of interest. Author David Camacho declares that he has no conflict of interest.

Acknowledgements

This work has been supported by the next research projects: EphemeCH (TIN2014-56494-C4-4-P) Spanish Ministry of Economy and Competitivity, CIBERDINE S2013/ICE-3095, both under the European Regional Development Fund FEDER, and SeMaMatch EP/K032623/1.

References

1. Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.
2. Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.
3. Zarni Aung and Win Zaw. Permission-based android malware detection. *International Journal of Scientific and Technology Research*, 2(3):228–234, 2013.
4. John Aycock. *Computer viruses and malware*, volume 22. Springer Science & Business Media, 2006.
5. Gema Bello-Orgaz, Jason J Jung, and David Camacho. Social big data: Recent achievements and new challenges. *Information Fusion*, 28:45–59, 2016.
6. Gema Bello-Orgaz, Héctor D Menéndez, and David Camacho. Adaptive k-means algorithm for overlapped graph clustering. *International journal of neural systems*, 22(05):1250018, 2012.
7. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

8. Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.
9. Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 48, 2007.
10. Takamasa Isohara, Keisuke Takemori, and Ayumu Kubota. Kernel-based behavior analysis for android malware detection. In *Computational Intelligence and Security (CIS), 2011 Seventh International Conference on*, pages 1011–1015. IEEE, 2011.
11. Byeongho Kang, BooJoong Kang, Jungtae Kim, and Eul Gyu Im. Android malware classification method: Dalvik bytecode frequency analysis. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, RACS '13, pages 349–350, New York, NY, USA, 2013. ACM.
12. Daniel T Larose. *Discovering knowledge in data: an introduction to data mining*. John Wiley & Sons, 2014.
13. Alejandro Martín, Héctor D. Menéndez, and David Camacho. String-based malware detection for android environments. In *Intelligent Distributed Computing X - Proceedings of the 10th International Symposium on Intelligent Distributed Computing - IDC'2016, Paris, France, October 2016. In press.*
14. Alejandro Martín, Héctor D. Menéndez, and David Camacho. Studying the influence of static api call for hiding malware. In *MAEB 2016 (XI Congreso Espaol de Metaheursticas, Algoritmos Evolutivos y Bioinspirados (MAEB 2016). In press.*
15. Alejandro Martín, Héctor D. Menéndez, and David Camacho. Genetic boosting classification for malware detection. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*. IEEE, 2016.
16. Mohd Zaki Mas' ud, Shahrin Sahib, Mohd Faizal Abdollah, Siti Rahayu Selamat, and Rubiyah Yusof. Analysis of features selection and machine learning classifier in android malware detection. In *Information Science and Applications (ICISA), 2014 International Conference on*, pages 1–5. IEEE, 2014.
17. Hector D Menendez, David F Barrero, and David Camacho. A genetic graph-based approach for partitional clustering. *International journal of neural systems*, 24(03):1430008, 2014.
18. David Meyer, Kurt Hornik, and Ingo Feinerer. Text mining infrastructure in r. *Journal of statistical software*, 25(5):1–54, 2008.
19. Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.
20. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.
21. Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.
22. J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European*, pages 141–147, Aug 2012.
23. Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.
24. Mohit Sharma, Meenu Chawla, and Jyoti Gajrani. A survey of android malware detection strategy and techniques. In *Proceedings of International Conference on ICT for Sustainable Development*, pages 39–51. Springer, 2016.
25. Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.
26. Guillermo Suarez-Tangil, Juan Tapiador, Flavio Lombardi, and Roberto Di Pietro. Alterdroid: Differential fault analysis of obfuscated smartphone malware.
27. Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2015.
28. Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.
29. Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.
30. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, May 2012.
31. Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.
32. Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm, 2001.