Expert Systems With Applications 95 (2018) 113–126

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Picking on the family: Disrupting android malware triage by forcing misclassification

Alejandro Calleja[a], Alejandro Martín[b], Héctor D. Menéndez[c,*], Juan Tapiador[a], David Clark[c]

[a] Department of Computer Science, Universidad Carlos III de Madrid, Madrid, Spain
[b] Departamento de Informática, Universidad Autónoma de Madrid, Madrid, Spain
[c] University College London (UCL), Gower Street, London WC1E 6BT, United Kingdom

ARTICLE INFO

Article history:
Received 31 March 2017
Revised 29 October 2017
Accepted 14 November 2017

Keywords:
Malware classification
Adversarial learning
Genetic algorithms
Iagodroid

ABSTRACT

Machine learning classification algorithms are widely applied to different malware analysis problems because of their proven abilities to learn from examples and perform relatively well with little human input. Use cases include the labelling of malicious samples according to families during triage of suspected malware. However, automated algorithms are vulnerable to attacks. An attacker could carefully manipulate the sample to force the algorithm to produce a particular output. In this paper we discuss one such attack on Android malware classifiers. We design and implement a prototype tool, called IagoDroid, that takes as input a malware sample and a target family, and modifies the sample to cause it to be classified as belonging to this family while preserving its original semantics. Our technique relies on a search process that generates variants of the original sample without modifying their semantics. We tested IagoDroid against RevealDroid, a recent, open source, Android malware classifier based on a variety of static features. IagoDroid successfully forces misclassification for 28 of the 29 representative malware families present in the DREBIN dataset. Remarkably, it does so by modifying just a single feature of the original malware. On average, it finds the first evasive sample in the first search iteration, and converges to a 100% evasive population within 4 iterations. Finally, we introduce RevealDroid*, a more robust classifier that implements several techniques proposed in other adversarial learning domains. Our experiments suggest that RevealDroid* can correctly detect up to 99% of the variants generated by IagoDroid.

1. Introduction

Detecting and classifying malware is a challenge that has steadily increased over time. Not only has the rate of production of distinct files been increasing but the methods used to evade detection have become more sophisticated. For instance, malicious apps have been observed colluding to achieve their desired outcomes (Labs, 2016; Zhou & Jiang, 2012). The quantity of malware targeting mobile devices doubled in the year to July 2016 (Labs, 2016), with a clear trend towards the reuse of source code instead of developing new variants from scratch (Zhou & Jiang, 2012). Mobile malware variants are produced through component reuse and also via obfuscation. Considering the advances in machine learning techniques in the last decades, there is widespread interest in applying these to the malware triage problem. Contemporary machine learning algorithms provide the potential to improve scalability and offer high flexibility regarding the features employed during the classification of malware into families (Dash et al., 2016; Gandotra, Bansal, & Sofat, 2014). However, an informed adversary can deliberately alter the decision process of an automated classifier by different means. The problem of employing machine learning algorithms in adversarial environments has previously been studied in security related contexts such as spam, intrusion detection, or malware classification (Biggio, Rieck et al., 2014; Dalvi, Domingos, Sanghai, & Verma, 2004; Lowd & Meek, 2005). In the same way, different countermeasures have been proposed (Biggio, Corona, Fumera, Giacinto, & Roli, 2011; Chinavle, Kolari, Oates, & Finin, 2009).

This paper investigates the automated disruption of Android malware triage, the process by which decisions are made in regard to the further analysis steps for a suspicious file (Chakradeo, Reaves, Traynor, & Enck, 2013). A critical step during this process, that may affect the choice of subsequent analysis techniques, is the identification of the malware family of a highly suspicious file. Our attack is that a malware writer, in deploying

* Corresponding author.
E-mail addresses: accortin@inf.uc3m.es (A. Calleja), alejandro.martin@uam.es (A. Martín), h.menendez@ucl.ac.uk (H.D. Menéndez), jestevez@inf.uc3m.es (J. Tapiador), david.clark@ucl.ac.uk (D. Clark).

https://doi.org/10.1016/j.eswa.2017.11.032

variants from a relatively novel family, attempts to disguise them as a different family, one that is less likely to attract intensive scrutiny. This may hide novel indicators of compromise such as DNS records, malicious URLs, or exploits (Lakhotia, Walenstein, Miles, & Singh, 2013).

In this scenario, the power of the malware writer or adversary is as follows: she has control over her malware sample and is able to extract static features such as intents-actions, API calls, and information flows. In addition, she knows the feature space used by the targeted classifier and has access to the classification/misclassification probability. This is a relatively strong assumption, yet the attacker still has the limitation of not knowing the underlying classification algorithm and she needs to preserve the semantics of the executable. Besides, she wants to automate the process. Our solution to this problem, a tool called IagoDroid, uses evolutionary algorithms to perform a search that identifies a minimal number of changes to the features in order to effect a family misclassification. IagoDroid can randomly choose a family or target a specific family.

Assuming further knowledge about the classifier is unrealistic in practice. Since the mapping (from vectors to labels) implemented by the classifier is unknown, there is no other option but to treat it as a black box that can be repeatedly queried during search. Even when this is not the case and the details about the classifier are fully known, obtaining an actionable analytical description of such a mapping might not be always possible, particularly for non-linear classifiers that capture complex interactions among features to produce the output label. Population-based search mechanisms such as genetic algorithms have proven to perform remarkably well in challenging domains where more traditional search algorithms have not succeeded (Sivanandam & Deepa, 2007).

Attacks against classifiers have been discussed before, both from a theoretical point of view and in particular security domains such as spam or intrusion detection. In this paper we study the impact of an attack against multiclass Android malware classifiers. Android apps are extremely easy to decompile, manipulate and repackage again into a new app. This makes it easy to introduce new artefacts (e.g., components, API calls, intents, information flows) in the app that will affect its associated feature vector and, therefore, the label given by a classifier. If carefully introduced (for instance, in if-then blocks only accessible through an opaque predicate that always evaluates to false), such modifications will not affect the app's execution semantics.

To demonstrate our approach, IagoDroid attacks family classification by RevealDroid (Garcia, Hammad, Pedrood, Bagheri-Khaligh, & Malek, 2015), a recently proposed malware classifier employing existing static analysis features. Our choice of RevealDroid is for convenience (it is open source and ready to use) and because it incorporates most of the static features discussed in the literature (API calls, information flows, and so on). However, IagoDroid is agnostic with respect to the classifier used and can be applied to different classifiers. Moreover, we have subsequently designed a countermeasure that can detect when a potential evasion has been performed and can recover a set of potential original families.

The main contributions of this paper are summarized as follows:

- We propose a novel classification evasion attack against any triage process where the family classification relies on static analysis. We demonstrate, in particular, that IagoDroid can evade an open source classifier named RevealDroid, a freely available multi-class malware classifier which combines several different features. To do so, we employ evolutionary algorithms, a technique which has been previously employed in the context of evading classifiers for security applications (Pastrana, Orfila, & Ribagorda, 2011; Xu, Qi, & Evans, 2016) (see Section 2).
- We train RevealDroid using 1919 malware samples from the DREBIN (Arp, Spreitzenbarth, Hubner, Gascon, & Rieck, 2014) dataset divided into 29 different malware families. IagoDroid successfully forces misclassification of 28 of the 29 families, in the process modifying only a single feature of the original malware feature vector. On average, IagoDroid is able to find the first evasive file within the first generation and converges on a 100% evasive population within 4 generations (see Section 4). It was able to find approximately 14,000 evasive variants from more than 290 initial malware samples within 2 min.
- The countermeasure, named by us as RevealDroid*, detects potential evasions in between 90% and 99% of the output of IagoDroid, depending on the number of modifications introduced, and can identify potential original families for the malware (see Section 5).

The rest of this paper is organized as follows: In Section 2, we present our approach, introducing issues related to our contribution such as the adopted adversarial model, the target classifier, and the parameters of the genetic algorithm component. Section 3 describes the experiments and our configuration of them. In Section 4 we analyse and discuss the results while Section 5 describes the countermeasure proposed. Section 6 introduces the most relevant, related contributions found in the literature and finally Section 7 concludes the paper.

2. IagoDroid

This section describes IagoDroid, a prototype tool that induces mislabelling of malware families during the triaging process for potential malware samples. Given the importance of automated systems to detect and classify malware, to understand how these systems can fail (and how can they be strengthened) when attacks are directed against their integrity is an important task. IagoDroid's main goal is to demonstrate that an attack on an Android malware classification tool is feasible, by forcing it to produce a family misclassification as the result of some minor changes in the original sample and without modifying its semantics.

Following the taxonomy of attacks on machine learning developed by Barreno, Nelson, Sears, Joseph, and Tygar (2006), our approach can be positioned as follows:

- **Exploratory Attacks:** The attack described in this paper is *exploratory* since it does not aim at altering the training process but the classification itself, offline.
- **Targeted Attacks:** Regarding specificity, the proposed attack is focused on misleading the label given by the classifier to a particular sample. Nevertheless, the use of evolutionary search to find a proper mutation strategy can be used to fool the detection of any sample in the dataset as demonstrated in the following sections of the paper.
- **Integrity Attacks:** In contrast to attacks against the availability of the classifier, we do not seek to induce random classification errors. We aim to coerce an intended family misclassification for specific input samples.

The basic idea behind the IagoDroid attack is that the feature vector of a malicious application can be transformed by injecting new specific, incremental values, and this can eventually result in the assignment of an incorrect family label. These changes in the feature vector require modifications in the app's code and resources, in order to build a new sample corresponding to the desired feature vector. For instance, it may be necessary to include a new API call. Moreover, these changes are made while simultaneously keeping the semantics of the app invariant. The

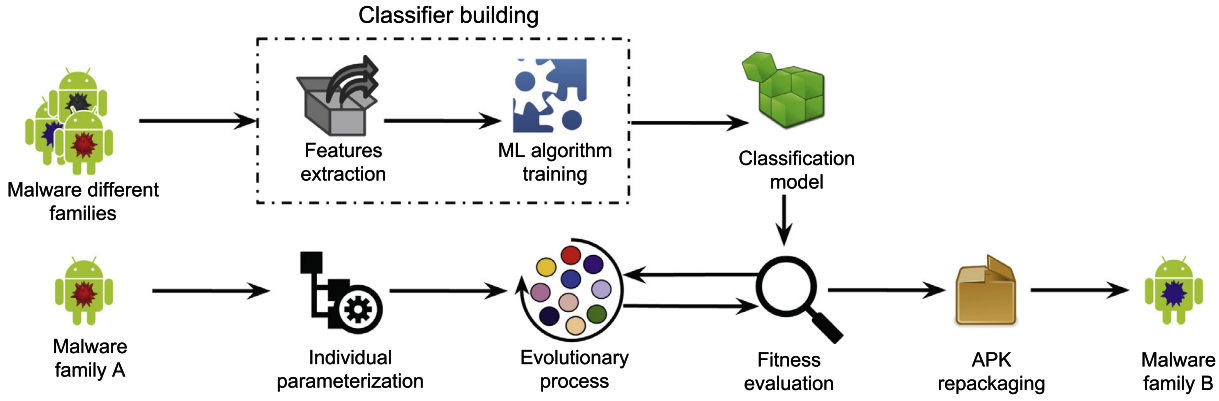

Fig. 1. General scheme of IagoDroid.

key to achieving semantic invariance is to only consider small, *incremental* changes in the app (i.e., adding new API calls or new permissions) each of which does not alter the original semantics.

Obtaining the list of transformations to apply to the feature vector can be seen as a search problem in which a search heuristic finds a solution (i.e., a new feature vector) based on the proximity of the current feature vector to one associated with a label different from the current one. This proximity can be calculated based on the output of a malware classifier, by measuring the probability of the feature vector being classified as the original label or as a different label.

Fig. 1 shows the general architecture of IagoDroid. There are two main pipelines, one depicted above the other. The upper pipeline shows the process of building the classification algorithm used to drive the heuristic search. This algorithm takes as input a set of samples placed in a feature space. These samples are employed to train the classifier and obtain a classification model. On the other hand, the process of performing the attack is shown in the bottom pipeline. In this case the process starts by picking a malware sample whose family label we wish to alter. Additionally, the attack pipeline can also take a target family (see Section 2.1). IagoDroid employs a genetic algorithm to perform the heuristic search, as these algorithms to adapt to problems of high complexity. The search is guided by a fitness function which uses the classification model previously trained to find the solutions that induce misclassification. Finally, the application is modified in order to adapt it to its new feature vector and it is repackaged to obtain a new app which is able to evade a correct family classification.

The following subsections present the context for IagoDroid, starting from a description of the adversarial model, a specification of the target classifier and finishing with the problem formalisation.

2.1. Adversarial model

In our scenario, we consider an adversary who aims to evade the correct classification of a sample belonging to family *A* by misclassifying it as family *B*.

The goal of the adversary is to ensure that it is possible to miss the identification of the correct family. We consider two cases. In the first scenario, the selection of the target family is delegated to the evolutionary algorithm which will merely try to change the label of the input feature vector with the minimum number of changes. In the second scenario, the target family is also an input and the search will attempt to find the feature changes that attain this specific misclassification.

As introduced in Section 1, the adversary seeks to thwart the deployment of proper countermeasures. To appreciate how the attacker achieves this, it is useful to specify what the adversary knows about the classifier. Given that IagoDroid is based on a well known classification algorithm whose source code is publicly available, we allow the feature set employed by the classifier to be known to the attacker. We assume the attacker is able to create new feature vectors and submit them directly to the classifier without any constraint. We assume that the classifier interacts with the submitted feature vectors as if they had been extracted from applications created or modified by the adversary. In other words, the search is conducted at the feature vector level, without directly modifying the malware sample until a solution is found.

Regarding the classifier output, the attacker receives two values: the label assigned to the input vector and a classification score, indicating the trust/reliability of the classification. Since the adversary is able to deploy her own implementation of the classifier, we do not consider any limitation in the number of feature vectors that can be submitted, hence the attacker has an unbounded number of attempts to lead the classifier to a compromised verdict.

This scenario for the adversarial capabilities is realistic since the target classifier can be well documented (i.e., no security through obscurity) or else reversed.

2.2. Target classifier

We decided to use an already proposed and documented classifier in our work. Our selection criteria for choosing a target classifier included good classifier precision and high diversity in the features it uses. While there are several classifiers discussed in the literature, few of them consider an important and representative set of features and are freely available to download. Table 1 compares the use of different features by the most important classifiers described in the literature and notes whether they can be downloaded to be used for our purposes. From the nine analysed proposals, only the authors of three of them have released the source code of their solutions, RevealDroid, Dendroid and DroidLegacy. Of these, RevealDroid is the most appropriate one since it uses the widest set of features. In addition, it was designed and tested for malware family classification.

RevealDroid classifier building

RevealDroid consists of a series of components that enable the extraction of three different kinds of data from Android apps: API calls, intent actions, and streams and flows. These features can be used to build a dataset and then to train a machine learning algorithm to perform a classification task that predicts the family label of previously unseen samples. Each group of features is extracted separately and is sequentially added to the feature vector for each application, with the objective of controlling and

Table 1
Android malware classification methods using machine learning approaches.

Classifier	Code structures	Permissions	Api Calls	Intent-actions	Flow analysis	Tested for families classification	Freely available to download
RevealDroid (Garcia et al., 2015)	✗	✗	√	√	√	√	√
DroidSIFT (Zhang et al., 2014)	✗	√	√	√	√	✗	✗
Dendroid (Suarez-Tangil et al., 2014)	√	✗	✗	✗	✗	√	√
Drebin (Arp et al., 2014)	✗	√	√	√	✗	√	✗
DroidMiner (Yang et al., 2014)	✗	✗	√	√	✗	√	✗
DroidAPIMiner (Aafer et al., 2013)	✗	✗	√	✗	✗	✗	✗
VILO (Lakhotia et al., 2013)	√	✗	✗	✗	✗	√	✗
DroidLegacy (Deshotels et al., 2014)	✗	✗	√	✗	✗	√	√
MAST (Chakradeo et al., 2013)	√	√	✗	√	✗	✗	✗

supervising the whole process. We used the original code of RevealDroid, downloaded from its public repository.[1]

The first feature extracted from each application is a list of the API calls found in the code, which allows one to obtain a high level description of the expected behaviour of the application. These API calls can be included in the feature vector of an app in two ways: grouping the calls by using the 30 security-sensitive API categories defined by Rasthofer, Arzt, and Bodden (2014), or grouping the calls by using the Android package in which they are defined. Revealdroid follows this second approach.

The second step of the dataset building process consists of including intent actions data. Intent actions are identifiers of different events that happen within the lifecycle of an application such as launching a new activity or a new service. This is also a useful information source for detecting and classifying malicious applications (Chin, Felt, Greenwood, & Wagner, 2011).

Thirdly, RevealDroid uses information flows to characterise the samples. An information flow can be seen as the path followed by a piece of sensitive data through the flow graph of a program. In this case, an information flow is represented as a pair consisting of a *source* (i.e. an API call providing data to the app) and a *sink* (i.e. the app providing data as input for another API call).

The final step involves the training process of a machine learning classification algorithm. The authors of RevealDroid use a decision tree based algorithm, C4.5, and the 1-nearest neighbour algorithm. Nevertheless, any other machine learning algorithm might be used instead.

2.3. Problem formalisation

In this subsection we provide a formal description of the attack.

Our experimental dataset can be formalised as the set X, containing samples of different malware families. However, since we are solely interested in the feature vectors describing different properties of each sample, X can be represented as the set of n feature vectors:

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

Each feature vector x_i is composed of k different features, extracted directly from the original application:

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

Initially, each sample in the dataset is labelled with the name of the family it belongs to. We name the set of all the possible labels in the dataset as Y. Thus, the classifier C can be defined as a function mapping a feature vector $x_i \in X$ to the most likely label $y_j \in Y$, paired with its probability of being the correct label:

$$C(x_i) = (p(y_j), y_j),\ \ y_j \in Y, \tag{3}$$

where $p(y_j)$ is the probability of y_j being the true label of x_i as estimated by the classifier.

[1] https://bitbucket.org/joshuaga/revealdroid.

Finally, we formalise our search approach at a high level of abstraction as a function accepting two arguments: a feature vector which is to be misclassified, obtained from the app, and the original label that we want to avoid. The output of this function (x_i') will be the original vector with a set of changes (e.g., increment the value of a feature) to be applied to the original feature vector x_i. Once this new vector has been created, the classifier C will assign a new label y_j' to this modified vector:

$$\text{IagoDroid}(x_i, y_j) = x_i' :\ \ y_j' \in Y,\ \ y_j' \neq y_j. \tag{4}$$

We consider the changes as a Δ vector satisfying: $x_i + \Delta = x_i'$.

2.4. Genetic approach

This section describes the design details of the genetic algorithm that is at the core of IagoDroid.

2.4.1. Encoding

Each individual I_i present in the evolutionary process is designed to represent a possible new feature vector x_i' containing k different features or genes. Since the goal of IagoDroid is to introduce modifications in the feature vector so that the associated app gets misclassified while preserving its semantics, the individual's encoding is designed to only allow incremental changes in each feature. Thus, the individual starts with the same feature vector as the sample received as input x_i. Once the minimum value of each gene I_i^j of the individuals is established, it is also necessary to fix a maximum threshold MT to limit the number of changes and facilitate their implementation. Then, $[x_i, x_i + MT]$ is the range for each gene in each individual I_i. This restriction on the values of each individual will be present through the entire evolutionary process.

2.4.2. Genetic operators

Four operators are in charge of driving the evolutionary process across a number of generations. The **selection** operator is elitist, picking the n best individuals in each generation to be part of the next generation. **Reproduction** is performed by means of a standard tournament operator. For **crossover** we opt for a uniform operator and, lastly, a random **mutation** operator is used to introduce diversity in the population by changing the value of some genes randomly (within the ranges specified above).

2.4.3. Fitness function

The fitness function uses the gradient of the classifier output (score) to guide the genetic search. Specifically it uses the probability of the class that the algorithm wants to avoid. This can be formally defined as:

$$f(x_i, y_j) = \begin{cases} 1 - p(y_j) & \text{if } (p(y_j), y_j) = C(x_i) \\ 1 & otherwise \end{cases} \tag{5}$$

where x_i is the feature vector of the application, y_j indicates its family and p represents the probability assigned to the classification.

2.5. Targeting specific families

The approach described so far addresses a genetic search seeking to reach different malware families, providing an effective technique to hide the real family of a malicious application. However, the search has no control over the final family label that will be assigned to the modified sample. This represents an interesting issue, since an attacker might well wish to target specific families with different purposes (for instance, to force defenders to deploy specific incorrect countermeasures). To address this issue, the fitness function can be easily modified to guide the search to individuals representing feature vectors classified as a given target family. The new fitness function is as follows:

$$f(x_i, y_k) = \begin{cases} p(y_k) & \text{if } C(x_i) \neq (p(y_k), y_k) \\ 1 & otherwise \end{cases} \quad (6)$$

where y_k represents the target family label.

3. Experimentation

We next discuss the experiments that we have performed to validate our proposal. The experiments address the following research questions:

- **RQ1:** *How much effort does it take to find the modifications needed to misclassify a particular sample?*
- **RQ2:** *Which features are more often involved when modifying a sample?*
- **RQ3:** *Given a malware family, is the cost of forcing misclassification errors in its samples constant for all possible target families or are some families easier to target than others?*

Our main goal is to provide evidence that our approach can induce a misclassification error in a targeted malware classifier. Accordingly, the experiments discussed in this section have been executed using only the first fitness function presented in Section 2.4.3 and we did not direct the genetic search towards a particular family classification. The rest of this section describes the experimental setting, including the dataset, classifiers and parameters used. To facilitate the reproducibility of our experiments, we have created open source versions of our implementation, dataset and scripts used throughout this work[2].

3.1. Dataset

We tested our approach using the DREBIN dataset (Arp et al., 2014). This dataset contains 5560 malicious Android apps classified into 179 different families. Unfortunately, the number of samples per family is not balanced, resulting in some families with a low number of samples (e.g., 47 families contain just 1 sample). We therefore removed all classes containing less than 10 samples, resulting in a final dataset composed of 5198 samples distributed in 54 different families.

We then leveraged a number of existing tools to extract the features from each sample in the dataset. API calls and intent actions were obtained using Androguard[3], a fairly well known static analysis tool. To extract information flows we used FlowDroid (Arzt et al., 2014), a taint analysis tool that finds *source-sink* connections. FlowDroid can be tuned through different parameters to maximise either performance or precision. We set parameters to achieve as much precision as possible. This approach differs slightly from the procedure followed by other works that have generally aimed at maximising performance by compromising precision (e.g., Garcia et al., 2015). Using FlowDroid to extract information flows introduces two important issues. First, the time it takes to analyse a single app ranges from a few minutes to several hours in the worst case. Furthermore, it unexpectedly crashes for many apps. These two issues (scalability and stability) forced us to dramatically reduce the number of samples actually used in the experiments. Thus, from the original set of 5189 samples, only 1919 samples belonging to 29 different families were successfully processed by FlowDroid.

Finally, once the final dataset was built, we carried out a basic covariance analysis among the features to remove those that did not provide any additional information.

3.2. Target classifier

To demonstrate our approach, we relied on RevealDroid, an Android malware classifier that uses various static features and allows the use of different machine learning classification algorithms. While the original authors used C4.5 and 1-NN, we restricted ourselves to C4.5 since it showed better accuracy and precision. Nevertheless, our approach is not limited to a particular classification algorithm and should work with any other classification approach. The C4.5 algorithm was trained using the RWeka package for R, keeping its default parameters. We use 2/3 of the data for training combined with 10 cross-fold validation and the remaining 1/3 for testing. The testing accuracy is 88% averaged over 50 runs.

3.3. Genetic search

The genetic algorithm was configured using the following parameters: a mutation probability of 0.1; a crossover probability of 0.8; population size equal to 50; maximum number of generations equal to 20; elitism parameter of 3; and a maximum number of transformations per allele of 1 (though we set an increment that provides a transformation probability per allele ranging from 0.6 to 1).

3.4. Attack steps

This subsection discusses the sequence of steps followed by an attacker to force the missclassification of a particular sample. Recall (Section 2.1) that we assume an adversary with full knowledge and unlimited access to the classifier.

The first step is to extract the features from the malicious samples that will be eventually mutated. Androguard and FlowDroid extract these features and generate the feature vector x_i. This feature vector provides a basis for the genetic search. Since the aim of the attacker is to change the final label of the sample without altering its functionality, the way in which the components of this vector may be modified during the search is restricted. For instance, if a particular API call is used in the original sample, the mutated sample must keep this feature (i.e., if the component of this API call is set to 1 in the original vector it cannot be set to 0). Otherwise the semantics of the application will be altered and the malicious behaviour will not be preserved. The genetic algorithm takes this into account and only mutates these features by adding additional intent actions, API calls or information flows, *without removing any of the original values*. Under this premise, the search process generates new individuals by evolving the previous generation. On each iteration, the fitness function evaluates for every single individual whether the correct classification has been evaded.

Once a solution is found, the attacker applies the mutation strategy found by the genetic search to the original malware sample. This will require adding a combination of new intent actions,

[2] The dataset is available at https://data.mendeley.com/datasets/4sksrpm5vj/1 and the code at https://github.com/hdg7/IagoDroid.

[3] https://github.com/androguard/androguard

API calls, and/or new information flows. To alter the original APK file, the attacker first decompresses it to access the files packed inside, such as the manifest or the DEX file(s). Adding a new intent action, API call or information flow requires disassembling the original DEX file, which contains the bytecode responsible for the app's functionality and is generated at compilation time from the original Java source code. There are several tools to carry out this process. Smali and Backsmali[4] are well known tools for translating the Dalvik bytecode contained the DEX file into human readable (smali) code. The result of disassembling a DEX file using these tools is a set of files related to the original Java sources. These files can easily be modified by the attacker to add a new call to an Android API method or a new intent action. To avoid introducing undesirable extra functionality into the app, the attacker can put the newly added code blocks within conditional sentences (i.e., if-then) driven by opaque predicates that always evaluate to false. This would prevent optimizers from removing them while achieving the two-fold goals of having those features in the code but not executing them. Once the new elements have been added to the code, the process can be reversed using Backsmali to repackage the APK file.

Unlike API calls or intent actions, information flows are related to the execution paths of the program. This means that a particular information flow will only be detected if it happens as part of the instructions that are actually executed when the app runs. This is a consequence of the way in which taint analysis tools based on symbolic execution, such as FlowDroid, explore the application to find possible data flows, building the application flow graph and following all the possible paths within the application. To insert a new information flow the attacker needs to place it within a method that will be eventually called. Android apps implement several callbacks (such as those used for managing the life-cycle of activities and services) to interact with different events taking place in the operating system. Thus, finding pieces of code that will certainly be executed is not difficult. To add a new information flow, the attacker can follow the procedure described above for intent actions and API calls.

We have manually tested the attacks with one of the samples in our dataset. Specifically, we modified an app labelled as a member of the Plankton family and, after altering it according to the found mutation strategy, the classifier misclassified it as a member of the BaseBridge family. Achieving misclassification only required the addition of a single intent action (ACTION INPUT METHOD CHANGED). After following the previously described steps, we examined the app and extracted the new feature vector. This new vector contains the feature ACTION INPUT METHOD CHANGED along with the original features of the app, showing that the modification step worked as expected while keeping the original features unchanged. Finally, we ran the classifier over this sample and obtained the wrong label (BaseBridge) as expected.

4. Results

We next discuss our experimental results. The experiments aim to provide answers to the three research questions introduced in the previous section. All the experiments were executed on a cluster of 6 nodes, each node equipped with 24 cores and 128Gb of RAM memory.

We took a random subset, selected uniformly across families, of samples from our main dataset for the experiments. This subset was composed of 290 samples, taking 10 samples per family from 29 different families. As we mentioned above, there are families that only have 10 samples, hence the need to pick at most 10 apps per family to balance the final sample.

4.1. Evasion effort

The first research question aims to measure the effort required by the attacker to find a mutation strategy that induces a classification error. We attempt to answer this question from three different points of view: (i) the number of generations required by the search to achieve evasion (i.e., to find a single individual evading the correct classification) and convergence (a whole generation evading correct classification); (ii) the number of modifications in the feature vector required; and (iii) the number of queries to the classifier (this is correlated with the first perspective but it is a standard metric in evasion environments (Biggio et al., 2013)).

Table 3 summarizes the results for the experiments carried out. It shows how many generations were enough to achieve evasion and at which point the genetic search converges (i.e., all individuals being misclassified). Remarkably, a solution is found in the first generation for all families but BaseBridge. This means that a single iteration of the genetic algorithm is required to evade the correct classification of a single sample. This achievement suggests that the search effort is low and the search might be replaced by a simple analytical process consisting on adding changes to the features (i.e., adding API calls, or intents among others). As a sanity check, we analysed this possibility considering transformations from a single sample of a specific family to another (in this case, from GinMaster to DroidKungFu). The analytical process can only add changes. However, all possible transformations from the vectors of GinMaster to vectors of DroidKungFu require subtractions. This would change the app semantics. Considering only those features that can be added, the analytical process requires between 500 and 16,000 changes from the original to the target vector. Using the same samples, the GA found solutions with only one change.

The average number of generations required to achieve convergence is around 4 for all families. Notable deviations include DroidKungFu, whose samples require around 7 generations, and SMSreg with less than 2 generations. This demonstrates that the evasion technique is extremely efficient against the classifier for the families tested. The only family whose samples cannot be successfully mutated so as to be classified as some other family is BaseBridge. A careful analysis of the results and the classifier's inner working for this family shows that samples with this label have a strong correlation with the ACTION_INPUT_METHOD_CHANGE feature. Every time this feature is present, the sample is classified as belonging to BaseBridge. Since the semantics preserving rules prevent us from removing any features, this poses a clear limitation on the attack.

The total amount of time taken for these experiments using the sample subset of 290 individuals is around 2 min. Within this time span, the search found 14,000 mutation strategies able to evade the classifier. This number can be broken down into 50 different mutation strategies for 280 individuals (omitting the ten individuals that belong to the BaseBridge family). This gives us interesting information about the performance of the attack and demonstrates how easy it is to evade a malware classifier such as RevealDroid.

To discover the minimum number of modifications needed to achieve misclassification, we set the change probability to the minimum value (0.6). The results are shown in Table 3. In this case, we selected malware samples uniformly from the whole dataset, considering a realistic scenario in which an attacker would employ different malware samples without any previous knowledge about their classifications. In this scenario, some samples were then misclassified by RevealDroid, showing that no modification is needed to evade it.

In almost all cases the average number of modifications is close to 1. This means that the evasion technique only needs to modify

[4] https://github.com/JesusFreke/smali

a single feature in order to evade classification, revealing how fragile malware classifiers such as RevealDroid are. In order to explore this case in more depth, Section 4.2 studies the impact of the different features during the misclassification attack.

Finally, the number of queries to the classifier during the evolution depends on the number of generations and the population size. In each generation a single query is executed for each sample in the population. Since our approach only needs one generation to succeed, 50 queries are needed per sample. This number is higher if we require the convergence of the whole population, as needs up to 350 queries (7×50) in the worst case.

RQ1. Our results show that misclassification can be achieved in just one generation of the genetic search. This translates to a number of queries to the classifier ranging from 50 to 350 per sample. Furthermore, only one mutation is needed to induce a misclassification error for most samples.

4.2. Relevant features for the attack

In order to understand which features are more related to a particular family, we performed an analysis of the most relevant features affected during the mutation process. We followed the same approach as in the experiments discussed in Section 4.1, where only one feature is needed to change the family classification. Table 3 shows the feature that is changed most often and the overall probability for this feature to be modified during the search.

The feature most frequently added is ACTION_INPUT_METHOD_CHANGED. As we mentioned above, this feature is tightly coupled with the BaseBridge family, in such a way that whenever it is added to the feature vector, the sample is classified as belonging to BaseBridge.

ACTION_USER_PRESENT is another feature that is present in the modifications, especially for families such as Kmin, Steek, Yzhc and Fatakr. These families are closely related to remote server connections (Kmin and Yzhc) and sending SMS messages (Steek and Fatakr) containing private information, so they do not necessarily focus on user actions.

RQ2. The feature ACTION_INPUT_METHOD_CHANGED is used most often due to its close relationship with the BaseBridge family. ACTION_USER_PRESENT is used next often, appearing in four families with a common behaviour (leaking information from the device).

4.3. Transition between families during evasion

The final experiment attempts to measure the difficulty of mutating samples from each family to each potential target family. To do this, we measured the most commonly changing patterns among the different families during the search. Fig. 2 depicts a probabilistic representation of the most frequent changes between families. Unsurprisingly, BaseBridge is the family to which samples are most commonly reclassified. This is related to our previous analysis in Section 4.2, which showed that any sample with the feature ACTION_INPUT_METHOD_CHANGED set to one is classified as belonging to BaseBridge regardless of any other features.

Some interesting relationships can be found, such as the one between Plankton and Nyleaker, which share almost the same intent actions, Plankton having a couple of actions more than Nyleaker. Kmin and GinMaster have a close relationship with

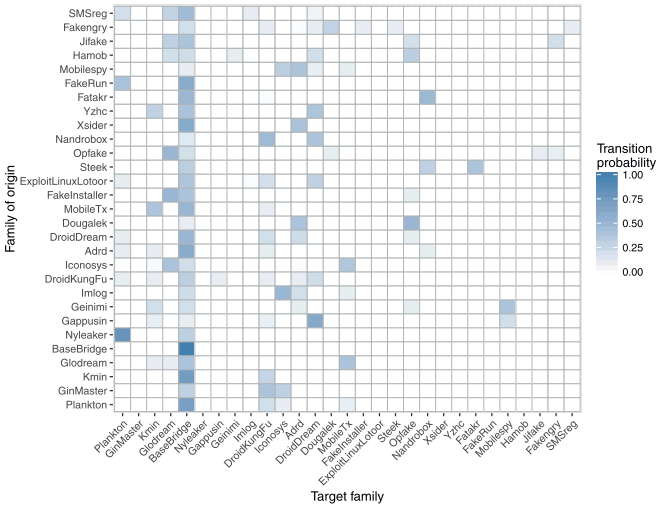

Fig. 2. Most frequent classification errors between families induced during the search.

DroidKungFu: a single modification of a flow based on MMS (Kmin) or the ACTION_USER_PRESENT feature (GinMaster) causes the original sample to be classified as belonging to DroidKungFu. A similar case happens with Fatakr and Nandrobox, in which a single modification of the ACTION_USER_PRESENT feature causes a misclassification.

Interestingly, the matrix shown in Fig. 2 is asymmetric. This means that samples from family A can be mutated into samples of family B but the inverse process was not found possible during the search. The only cases in which both mutations are possible are Plankton and DroidKungFu, DroidKungFu and Kmin, and Adrd and DroidKungFu. This suggests that DroidKungFu is a heterogeneous family. Finally, we note that there are 9 families that can never be targets: GinMaster, Nyleaker, Geinimi, Imlog, ExploitLinuxLotoor, Xsider, Yzhc, FakeRun and Hamob. This is a consequence of how the classifier builds the classification model, keeping some families bounded to specific feature ranges that are modified during the evolution process.

RQ3. The effort required to mutate a sample from an original classification to classification as a target family depends on both families, with some mutations being impossible.

5. A countermeasure

The results discussed in the previous section demonstrate that it is generally possible (and in fact easy) to cause a misclassification error in a typical Android malware classifier. This is ultimately accomplished by injecting additional artefacts into the sample, such as new API calls or intents, that will affect the feature vector associated with the app.

We next discuss how such attacks can be countered through the use of a more robust classifier. Our proposal first aims at detecting potential attack cases (i.e., samples deliberately modified so as to induce a classification error) and then at backtracking the changes to identify potential source families. Both strategies constitute variations of ideas proposed before in the field of adversarial machine learning (Chinavle et al., 2009). However, this is the first countermeasure discussing the ability to backtrack the attack.

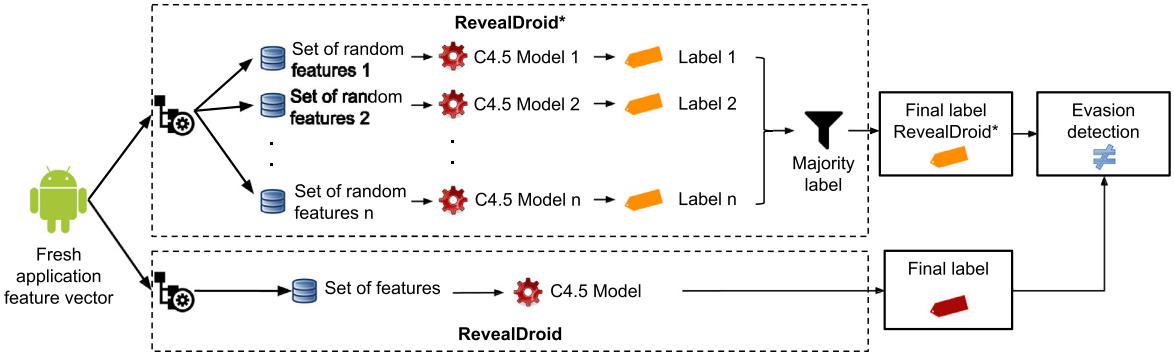

Fig. 3. Countermeasure schema including RevealDroid*.

5.1. Detecting potential misclassifications

The result of an attack, such as the one shown in this paper, is an app modified in a way that will deceive a classification system, causing it to return an incorrect family label. The underlying causes for such an error are related to the manner in which the algorithm at the core of the classifier works. In the case of RevealDroid, each alteration introduced in the application translates into features that will change the path followed along the decision tree, thus driving the output to a different leaf and, therefore, a different label.

In order to detect potential attempts to evade the classifier, we propose an extension of the target classifier: Revealdroid*. This enhanced version of RevealDroid employs a pool of C4.5 trees instead of relying on just one instance. Each classifier in the pool makes decisions based on different subsets of features present in the feature vector, making it more robust against deliberate modifications. Thus, each classifier chooses its own subset of features randomly at runtime. Therefore, a potential attacker has no evident way of modifying the vector in order to evade all the classifiers at once. The final label assigned to a sample by this enhanced version of RevealDroid results from majority voting. Our countermeasure is inspired by those proposed by different authors in the literature. The *bagging* (boosting and aggregating) approach has proven to be effective in enhancing the robustness of classifiers in various related problems (Biggio et al., 2011; Perdisci, Gu, & Lee, 2006).

Fig. 3 shows the architecture proposed for RevealDroid* and the whole schema for the countermeasure. The countermeasure consists of measuring the level of agreement between RevealDroid and RevealDroid*. When these two tools disagree, we consider that an attacker achieved a potential evasion. RevealDroid* must keep the same classifier, training data and parameters as RevealDroid, in order to generate similar outputs and reduce the false alarm (or false positive) rate. However, for the triage process, the priority is to reduce false negatives in order to guarantee that an important sample is not misclassified as irrelevant.

The feature extraction process of RevealDroid* remains unchanged, using the same feature vector for each app with a list of API calls, intent actions and information flows. Once the feature vector is generated for each app, features are randomly partitioned into a number of groups. That is, each feature is randomly assigned to one (and just one) group, guaranteeing that all groups have the same number of features. The number of groups can be manually tuned and also equals the number of classifiers (C4.5 in our case) used in the ensemble. Each classifier is then trained with all the instances using the subset of features allocated for it, seeking to maximise the separation among labels in this reduced feature space.

The classification process for a new malware sample with RevealDroid* is also outlined in Fig. 3. Once again, the feature vector is generated following the rules of RevealDroid. In a second step, the list of features is divided into groups depending on the split previously performed when training the models. Each instance of the C4.5 algorithm delivers a label according to its portion of the feature space and a majority rule is applied to obtain the final label for the input sample.

The strength of RevealDroid* lies in reducing the fragility of a single-classifier structure such as that of RevealDroid, in which just a simple change in the feature vector may lead to a classification error. When using multiple classifiers, the effort required to achieve a successful evasion becomes considerably more complex since the attacker needs to evade the majority of the classifiers in the ensemble. As a sanity check on RevealDroid*'s classification ability, we calculated its accuracy (see the plot at the bottom of Fig. 6). The accuracy (88%) is similar to that of RevealDroid (75–91%).

To evaluate the ability of the countermeasure to detect when a sample has been altered so as to evade a correct classification, we have used our attack to generate a representative set of apps successfully mutated, departing from, and trying to reach, all possible families following the approach described in Section 2.5. With this procedure, a subset of more than 10,000 individuals were successfully mutated. All these individuals were classified using RevealDroid*, yielding the results showed in Fig. 4. Each series in the figure is related to a specific configuration of the genetic algorithm, where an increment of 1 means that it is possible to generate individuals with up to 30 changes in the feature vector, whereas an increment of 0.6 reduces the number of changes to around 1. The reasons for this relationship between the increment parameter and the number of possible changes lies in the probability used internally by the genetic algorithm. Since every change must be manually injected into the application by the attacker, we may assume that in most cases the attacker would be interested in applying the minimum number of changes needed to achieve misclassification as a different family. This situation is represented by an increment equal to 0.6. In contrast, if the number of changes is not an issue for the attacker, a higher value of this parameter can be considered. As Fig. 4 shows, the label delivered by RevealDroid* differs considerably from the fake label pursued by the attacker, thereby notifying of a potential classification attack. With a maximum increment of 0.6 (around 1, 2 or 3 changes injected), using 14 different classifiers RevealDroid* will fail to detect the attack in 0.9% of the cases (false negatives), which means that the evasion will be detected in 99.1% of the evaluations.

The false positives of the countermeasure are computed by evaluating RevealDroid* with RevealDroid's test data (this data has non-mutated fresh samples for RevealDroid and RevealDroid*). The level of disagreement is calculated using the test output. Fig. 6 (top) shows that the false positive rate of our countermeasure ranges between 8% and 13.5%. Because the focus

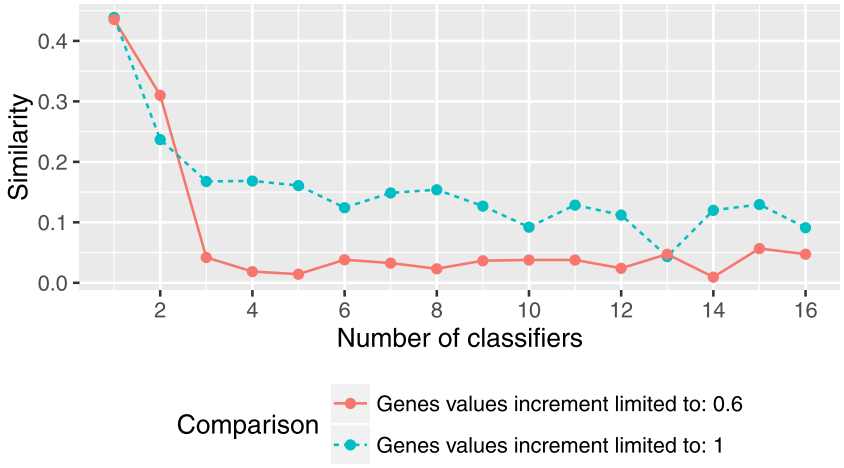

Fig. 4. False negative rates for the countermeasure with respect to the number of classifiers used in RevealDroid*.

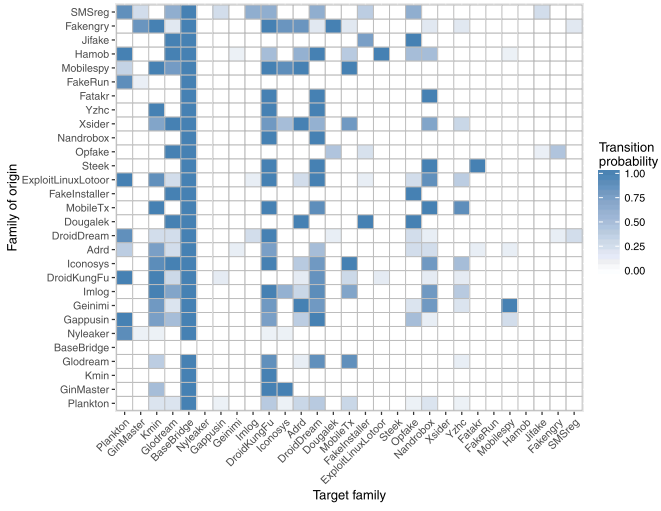

Fig. 5. Transition matrix to target families.

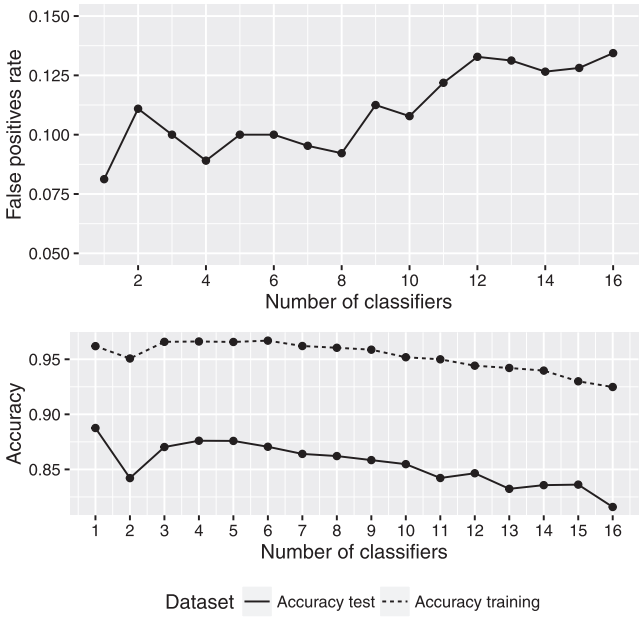

Fig. 6. False positive rate for the countermeasure (top) and accuracy of RevealDroid* depending on the number of classifiers used in RevealDroid* construction.

of the triage process is avoiding false negatives, we consider this result reasonable.

5.2. Reversing the attack

Once a sample has been suspected as the result of a misclassification attack, determining its original family is the next natural step. Reversing the transformation process that the attacker may have implemented is a complex task, particularly because of the difficulty of differentiating between the original app behaviour and the actions deliberately injected to cause the classification error.

However, the search process used during the attack offers the means to evaluate a number of possible original family classification candidates. The search was used between each possible pair of families in order to evaluate the transition probabilities between them (as the number of individuals belonging to a specific family able to reach a target family divided by the total number of individuals in the original family). The results of this experiment are shown in Fig. 5. The fitness function used here is the one described in Section 2.5, which allows one to target specific families. Since this matrix represents all possible transitions between original malware samples of different families and mutated samples, it is also possible to use this artefact to reveal the possible source families of an application detected as misclassified. Furthermore, it is also possible to order these candidate families by the transition probabilities.

For instance, Fig. 7 shows the probabilities of being the original family of a malware sample classified as Kmin family, according to the corresponding row of Fig. 5. In this example, there are 6 potential source families in which all the individuals were successfully mutated to be classified as Kmin, and these form a set of 6 prospective original families.

6. Related work

In this section we discuss the context for our work as it relates to Android static analysis, adversarial machine learning and countermeasures.

6.1. Android static analysis

Our work is focused on attacking a machine learning algorithm which operates on a space generated by static analysis

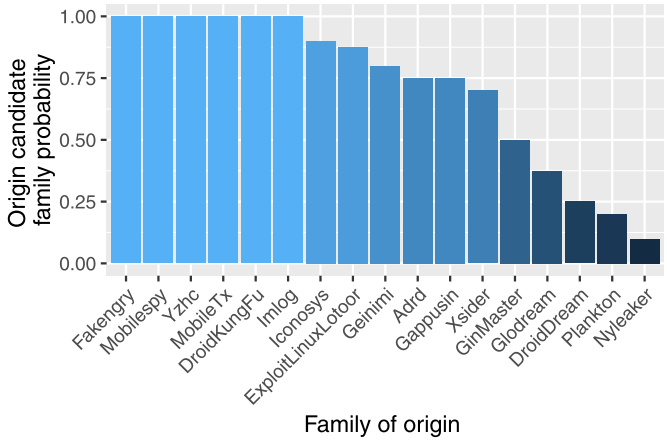

Fig. 7. Probabilities of being the origin family of a malware sample mutated to Kmin family.

features. We discuss the most relevant static analyses for our work: permissions, API calls, intent actions and flows.

Permissions have been identified as potential signifiers of malicious intentions. Tools like Kirim (Enck, Ongtang, & McDaniel, 2009) were used to detect anomalous settings containing malicious behaviour and tools like DroidRanger (Zhou, Wang, Zhou, & Jiang, 2012) leverage heuristics to perform the same task. API calls can be used to detect malware and generate signatures, which is the case for DroidLegacy (Deshotels, Notani, & Lakhotia, 2014) and DroidAPIMiner (Aafer, Du, & Yin, 2013). Current trends use both flow analysis, i.e. information leaks between data sources and potentially malicious sinks, and intent actions, remote procedures where one application can use the privileges of another one to perform malicious activities. Flows have been studied using tools such as FlowDroid (Arzt et al., 2014) and DroidSafe (Gordon et al., 2015), while intents have been studied in different ways: from the detection of communication vulnerabilities using ComDroid (Chin et al., 2011); to validation of the interaction between components with Epicc (Octeau et al., 2013); to points to communication between objects in different applications using Amandroid (Wei, Roy, & Ou, 2014); and to hybridization these methods, as seen in DidFail (Klieber, Flynn, Bhosale, Jia, & Bauer, 2014) which hybridizes Epicc and FlowDroid to improve detection through aggregated information.

Other work, out of the scope of our analysis but also related to static analysis for Android, uses a description language to identify semantic-based signatures, such as Apposcopy (Feng, Anand, Dillig, & Aiken, 2014), or aims to detect the context that triggers the malicious behaviour, such as AppContext (Yang et al., 2015) and TriggerScope (Fratantonio et al., 2016).

In our work, we target techniques that use static analysis features and leverage machine learning algorithms to detect or classify malware. These techniques, provided in Table 1, use the previously discussed tools to extract feature vectors that feed a machine learning algorithm. Tools like DroidSIFT (Zhang, Duan, Yin, & Zhao, 2014) and DroidAPIMiner (Aafer et al., 2013) have only been used for the detection problem, in which malware and goodware must be discriminated, while tools like Dendroid (Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014), DroidLegacy (Deshotels et al., 2014), Drebin (Arp et al., 2014), DroidMiner (Yang, Xu, Gu, Yegneswaran, & Porras, 2014) and RevealDroid (Garcia et al., 2015) have also been used for family classification, with RevealDroid covering the largest spectrum in the feature space. This was the main reason for choosing RevealDroid as the targeted classifier in our work. We also targeted the triage problem, which is closely related to the family classification problem as Lakhotia et al. state during the description of their tool VILO (Lakhotia et al., 2013). This problem has also been examined from a detection perspective using ranking based algorithms in MAST (Chakradeo et al., 2013). Our goal here was to attack the triage process using adversarial machine learning.

6.2. Adversarial machine learning

Evasion and Adversarial Learning (Huang, Joseph, Nelson, Rubinstein, & Tygar, 2011) are widely studied topics in both the machine learning and computer security areas (Barreno et al., 2006; Lowd & Meek, 2005; Ptacek & Newsham, 1998). Given the success of machine learning techniques for addressing security related problems such as malware analysis, spam identification, or intrusion detection, testing the resilience and robustness of these approaches against an informed adversary is a necessary activity.

There is a wide spectrum of applications of machine learning algorithms in classification problems. Their reliability is closely linked to the reliability of the systems that depend on them. Adversarial learning is then an important problem that must be addressed. According to Barreno et al., the main weaknesses of machine learning algorithms lie precisely in their adaptation ability, which can be exploited by attackers to cause deliberate errors (Barreno, Nelson, Joseph, & Tygar, 2010). This presents a complex issue, since machine learning theory takes as its basis that the training dataset used in a learning process remains representative of the problem domain and assumes intentionally harmful modifications of the data do not happen (Laskov & Lippmann, 2010).

The problem of learning in hostile environments was first considered by Kearns and Li (1993). In this work, the authors developed an extension to Valiant's *Probably Approximately Correct* (PAC) framework (Valiant, 1984; 1985). The extension allows the algorithm to learn even when a dataset has been polluted with erroneous data, introduced by an active adversary. This adversarial behaviour is modelled following a worst-case approach (i.e., unbounded computational power and access to the classification history are assumed). The main contribution of this work was to provide methods to limit the maximum portion of the dataset polluted by the adversary without having a negative effect on the classification result.

The proliferation of classification and detection tools relying on machine learning techniques has promoted an increased interest in attacking these tools, taking advantage of the weaknesses in classification algorithms. These attacks are very varied and depend mainly on the adversarial model considered, since the capabilities of the attacker and her knowledge about the classifier define the impact of the attack.

All these attacks against machine learning can be categorised by point of view. From a coarse perspective, the attacks can be classified in two categories: poisoning attacks (Biggio, Nelson, & Laskov, 2012) and evasion attacks (Xu et al., 2016). In the former case, the attack is performed during the training stage. In this scenario the adversary introduces fake or malformed data into the training set. This will lead the classifier to learn an inaccurate model and then classify further instances incorrectly. In the latter case, the attack is performed during the classification stage. The feature vector belonging to a particular sample is modified so as to force the classifier to produce a wrong label. The proposed attack in this paper falls in the evasion category as we try to fool an already trained model by distorting the feature vector of a particular sample.

Barreno et al. (2006) provide an extended taxonomy of the different attacks against machine learning applications. They model attack spaces using three key concepts: influence (*whether it affects the training stage or the classification itself*), specificity (*whether the*

attack tries to misdirect the classification of data belonging to a particular class or, alternatively, causes no discrimination to happen) and the security property violated by the attacker (*whether the attack is against the classifier's availability or against the result's integrity*).

A practical example of how classification algorithms can be successfully evaded is the classifier-agnostic attack strategy described by Biggio et al. for assessing the security of machine learning applications (Biggio et al., 2013). They propose an adversarial strategy based on gradient descent attacks. They consider different threat models depending on how much information the attacker has regarding the attacked classifier. The authors demonstrate how their strategy could be employed to evade classifiers such as SVM and neural networks trained for detecting malicious PDF files.

In an approach similar to our own work but applied to a different problem, Vigna, Robertson, and Balzarotti (2004) developed a framework for measuring the resilience of a signature-based Network intrusion detection system (NIDS) against an adversary. The authors employed mutation strategies for modifying known exploits. Mutations were applied at network, application and code level, and included modifying the shape of network packets, injecting malformed data, and hiding malicious code using polymorphic engines. Ten real world exploits were mutated using different strategies and used to measure the resilience of two NIDS products. The experiment confirmed that evading the NIDS signatures is feasible, especially when combining different mutation techniques. Although this research has the injection of specific information to provoke a malfunction in a detection system in common with the attack that we describe in this paper, the problem varies significantly, since we are focused on malware classification.

Other research focused on NIDS was presented by Pastrana et al. (2011). It also takes advantage of genetic programming, in this case as a search heuristic for finding modification routines capable of evading a particular NIDS. These modification routines take a malicious network packet as input and apply different adjustments (e.g.: changing a particular value within the data payload, altering the TCP header, etc.). The authors tested the framework against C4.5 and Naïve–Bayes. By using this genetic search, the authors obtained individuals able of inducing non-negligible error rates in both classifiers, attaining a 37% classification error rate in the Naïve–Bayes classifier. In our work, by contrast, we are finding modifications to evade a correct family malware classification, rather than evading its detection (thus assuming that the sample will still be detected as malware with high probability).

Genetic programming has also been successfully employed in fooling the detection of malicious code. In particular Xu et al. presented EvadeML, a framework for automatically evading PDF malware classifiers (Xu et al., 2016). PDF files have been frequently used by attackers as hosts for embedded malware. The authors of that paper employed genetic search for finding the best modification strategy leading to evasion of detection by two PDF malware detection systems (PDFrate and Hidost) built on top of machine learning solutions. Up to 500 malicious payloads were successfully evaded using the discovered strategies. Again, an evolutionary algorithm is used to evade detection rather than to evade a correct classification between malware families.

Another example of adversarial learning to evade the detection of malicious PDF files is the *mimicry attack* (Maiorca, Corona, & Giacinto, 2013) that injects malicious code into a benign file using 3 different strategies: injecting an EXE (EXEembed), a PDF (PDFembed) or a Javascript (JSinject) payload. The evaluation of these tools shows high detection evasion effectiveness (100% for EXEembed and PDFembed and 80% for JSinject) on the 6 variants generated by the authors. Again, in contrast to our attack, this research is not focused on re-shaping malware for evading a correct classification and the domain is different. The evasion of PDF detection has been extensively analysed in Laskov (2014), demonstrating the vulnerabilities of a known online PDF analyser to this kind of attacks. The interaction between malware families has indeed been studied but from an unsupervised learning perspective, using clustering algorithms (Biggio, Rieck et al., 2014). Here, the authors inject new samples into the training process with the aim of disrupting the result.

On the Android side, there are new evasion strategies which aim to attack machine learning (Grosse, Papernot, Manoharan, Backes, & McDaniel, 2016; Meng et al., 2016) and antivirus systems (Aydogan & Sen, 2015; Meng et al., 2016; Xue et al., 2017; Zheng, Lee, & Lui, 2012). The first technique in this area was ADAM (Zheng et al., 2012), which manipulates malware via re-packing and obfuscation. ADAM was created to audit antivirus systems and it showed good effectiveness against VirusTotal, reaching an evasion rate close to a 50%. In a similar line, Aydogan and Sen (2015) include a genetic programming framework to the obfuscation process, reporting an evasion effectiveness up to 33% against 8 antivirus systems. The effectiveness of evasion strategies was extended to new machine learning techniques, such as deep learning, by Grosse et al. (2016), who were able to reach up to an 80% evasion rate by adding perturbations to the malware variants through junk code. This effective strategy was also followed by Mystique (Meng et al., 2016), and its extension, Mystique-S (Xue et al., 2017). The former uses a multi-objective genetic algorithm to reduce the classification rate of machine learning algorithms and anti-viruses, while it maximizes the attack behaviour; the later generates the code dynamically to reach the same goal. They are able to evade the detectors up to 80% of the time for Mystique and 94% of the time for Mystique-S. An interesting case for evasion, out-of-the-box from the previous techniques, was introduced by Vidas and Christin (2014) who generated an attack based on red pills, i.e., detecting environmental conditions. This strategy combines the detection of behaviour, performance, hardware and software components. They were able to reach an 86% evasion rate. Our tool, IagoDroid, is focused on attacking the static analysis features of a machine learning classifier, and it is able to reach a 97% evasion rate. Compared with the other related tools of the state of the art, it is a competitive result (for a summary comparing all the above mentioned tools, see Table 2.

6.3. Counteracting adversarial learning techniques

The security community has worked both on testing classification systems built on top of machine learning techniques against different kinds of attacks and on designing countermeasures to deal with this problem. For instance, Chinavle et al. studied the effect of employing learning ensembles for combatting adversaries in a spam detection scenario (Chinavle et al., 2009). Their approach demonstrated that through the use of different classifiers, it is possible to detect performance degradation (due to evasion attacks on behalf of a motivated adversary) and automatically repair this condition. Their approach allows the system to maintain a high degree of accuracy through time while reducing the number of re-training stages.

Barreno et al. elaborated on the security and reliability of machine learning (Barreno et al., 2006), proposing a framework to evaluate the security of a particular machine learning application. In the same line, Biggio, Fumera, and Roli (2014) proposed a framework to introduce countermeasures against attackers while designing the classifier, instead of applying them later during training or test stages. The main contribution of this work is the lack of bounds for a particular classifier, making the framework flexible.

Table 2
A comparison among different evasion methodologies related to IagoDroid, separated in general techniques and Android specific.

Method	Target	Type of attack	Evasion rate
PDF and Network			
Pastrana (Pastrana et al., 2011)	C4.5 & Naïve Bayes	Network injection	37%
Vigna (Vigna et al., 2004)	Network intrusion detectors	Mutation of exploits	90%
Biggio (Biggio et al., 2013)	SVM & Nearest Neighbour	Noise injection & Gradient Descent	up to 100%
EvadeMl (Xu et al., 2016)	PDFRate & Hidost	Genetic Programming	90%
EXEembed (Maiorca et al., 2013)	PDF malware detectors	EXE payload embedding	up to 100%
PDFembed (Maiorca et al., 2013)	PDF malware detectors	PDF embedding	up to 100%
JSinject (Maiorca et al., 2013)	PDF malware detectors	Javascript embedding	up to 80%
Laskov (Laskov, 2014)	PDFRate	Noise injection	up to 72%
Biggio (Biggio, Rieck et al., 2014)	Behavioural Clustering	Data poisoning	76%
Android			
Mystique (Meng et al., 2016)	Anti-virus & Machine Learning	Code injection & Genetic Algorithms	up to 80%
Mystique-S (Xue et al., 2017)	Anti-virus	Dynamic code generation	94%
Vidas (Vidas & Christin, 2014)	Dynamic Analysis tools	Red Pills	86%
Grosse (Grosse et al., 2016)	Deep Learning	Perturbation	up to 80%
ADAM (Zheng et al., 2012)	Anti-virus	Re-packing and obfuscation	up to 50%
Aydogan (Aydogan & Sen, 2015)	Anti-virus	Genetic Programming and Obfuscation	up to 33%
IagoDroid	RevealDroid	Code injection & Genetic Algorithms	97%

Table 3
Summary of experimental results. The table shows, for each malware family, the number of generations required to find a first solution, the average number of generations required to achieve convergence, the average number of modifications, and the feature that is most frequently changed.

Family	First sol.	Avg. conv.	Avg. mod.	Feature
Plankton	1	3.3	1.0	ACTION_INPUT_METHOD_CHANGED (0.7)
GinMaster	1	3.7	1.0	SMS_MMS (0.6)
Kmin	1	4.3	1.0	ACTION_USER_PRESENT (0.6)
Glodream	1	4.7	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
BaseBridge	Inf	Inf	–	–
Nyleaker	1	3.6	1.0	NETWORK__LOG (0.4)
Gappusin	1	3.4	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)
Geinimi	1	3.9	1.0	NETWORK_INFORMATION (0.5)
Imlog	1	4.7	1.2	ACTION_INPUT_METHOD_CHANGED (0.7)
DroidKungFu	1	7.2	0.7	IPC__NETWORK (0.2)
Iconosys	1	3.5	1.1	NETWORK__LOG (0.3)
Adrd	1	3.6	0.8	ACTION_INPUT_METHOD_CHANGED (0.5)
DroidDream	1	4.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Dougalek	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
MobileTx	1	3.2	1.0	FILE (0.5)
FakeInstaller	1	3.5	1.0	ACTION_INPUT_METHOD_CHANGED (0.5)
ExploitLinuxLotoor	1	2.1	0.8	ACTION_INPUT_METHOD_CHANGED (0.4)
Steek	1	3.9	1.0	ACTION_USER_PRESENT (0.4)
Opfake	1	4.8	0.9	ACTION_INPUT_METHOD_CHANGED (0.5)
Nandrobox	1	3.2	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Xsider	1	3.1	1.0	ACTION_INPUT_METHOD_CHANGED (0.6)
Yzhc	1	4.5	0.8	ACTION_USER_PRESENT (0.4)
Fatakr	1	3.2	1.0	ACTION_USER_PRESENT (0.7)
FakeRun	1	4.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.4)
Mobilespy	1	3.1	0.9	ACTION_MAIN (0.4)
Hamob	1	3.4	1.0	ACTION_INPUT_METHOD_CHANGED (0.3)
Jifake	1	2.6	0.8	android.net (0.3)
Fakengry	1	2.6	0.6	UNIQUE_IDENTIFIER_DB_INFORMATION (0.2)
SMSreg	1	1.6	0.9	ACTION_INPUT_METHOD_CHANGED (0.3)

Dalvi et al. (2004) studied the development of robust classifiers. They addressed the problem as a game between the attacker and the target classifier. In their approach, the attacker's strategy is used as input for generating a classifier resilient to particular adversarial behaviour. Addressing the problem from a game theoretical perspective, the authors improved the working of vanilla Naïve–Bayes classifier in a spam detection case, dramatically reducing the number of errors.

From a more general point of view, the effectiveness of different strategies that deal with evasion attacks has been studied elsewhere. For instance, Support Vector Machines have been evaluated (Russu, Demontis, Biggio, Fumera, & Roli, 2016), concluding that the selection of the kernel function is crucial. Feature selection based countermeasures have been studied (Budhraja & Oates, 2015), showing that this can be counterproductive since it reduces the accuracy of the classifier in some cases. There is also a framework focused on evaluating the potential attack scenarios due to the use of feature selection methods (Xiao et al., 2015).

Although the above countermeasures are able to successfully narrow the effects produced by attacks on machine learning classifiers, they are mainly focused on detection problems: a binary classification between benign and malicious software. However, a classification task into different malware families constitutes a different scenario in which there can be an important number of classes closely located in the search space.

7. Conclusions

IagoDroid demonstrates that any Android malware classification scheme that relies exclusively on static analysis during triage is a sensitive process that can easily be destabilised. IagoDroid is able to fool the RevealDroid classifier into misclassifying the family for

28 out of 29 families in the dataset by modifying a single feature of the original malware. In the process, this attack generates up to 14,000 new variants for 290 malware samples in just 2 min.

As a countermeasure, we split the feature space into different overlapping sets where different classifiers work together to detect potential evasions. This method, named RevealDroid*, is demonstrably effective for a small number of modifications but less useful when the number of modifications is high. In the latter case, it is able to reduce the number of evasive variants that IagoDroid generates, but cannot prevent it from generating at least some. In consequence, RevealDroid* forces producers of malware variants to find techniques to modify a higher number of features during the variants generation process. In the case where an evasive file is detected, our countermeasure is also able to track the original malware family, providing an opportunity to reconsider the malware priority during the triage process.

This countermeasure shows some of the limitations of the evasion method. The first limitation is related to the adversarial environment. IagoDroid has full knowledge of the underlying classifier. This limits the possibility of having strong results in different scenarios, even when the technique may still be applicable. Another significant limitation of the technique is in the final generation of the variants, which in the current version requires human intervention to transform the suggested vector of changes into the actual variant.

These limitations inspire several, possible lines of future work, starting from measuring the ability of IagoDroid to cause misclassification in commercial tools, such as antivirus engines. This line of research would require an extension to the tool's capabilities such as providing automatic injection of changes within opaque predicates. From a research perspective, IagoDroid is useful for studying the limitations on the robustness of machine learning classifiers and, indeed, our future work will focus on defining sound measures based on evasion abilities. This will help to understand which classifications algorithms are stronger than others when faced with adversaries for algorithms based on both static and dynamic analysis. Finally, the backtracking ability of our countermeasure can be understood as a Markov model among families and transitions. This knowledge can be used to study more deeply the different relationships among Android malware families.

Acknowledgements

This work has been supported by the following grants: EphemeCH (MINECO TIN2014-56494-C4-4-P) and CIBERDINE (CM S2013/ICE-3095), both under the European Regional Development Fund FEDER; SeMaMatch EP/K032623/1 and InfoTestSS EP/P006116/1 from EPSRC; SPINY (MINECO TIN2013-46469-R) and SMOG-DEV (MINECO TIN2016-79095-C2-2-R) and Justice Programme of the European Union (2014-2020) 723180 – RiskTrack – JUST-2015-JCOO-AG/JUST-2015-JCOO-AG-1. The contents of this publication are the sole responsibility of their authors and can in no way be taken to reflect the views of the European Commission.

References

Aafer, Y., Du, W., & Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems* (pp. 86–103). Springer.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., & Rieck, K. (2014). Drebin: Effective and explainable detection of android malware in your pocket.. *Ndss*.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., et al. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices, 49*(6), 259–269.

Aydogan, E., & Sen, S. (2015). Automatic generation of mobile malwares using genetic programming. In *European conference on the applications of evolutionary computation* (pp. 745–756). Springer.

Barreno, M., Nelson, B., Joseph, A. D., & Tygar, J. D. (2010). The security of machine learning. *Machine Learning, 81*(2), 121–148. doi:10.1007/s10994-010-5188-5.

Barreno, M., Nelson, B., Sears, R., Joseph, A. D., & Tygar, J. D. (2006). Can machine learning be secure? In *Proceedings of the 2006 ACM symposium on information, computer and communications security* (pp. 16–25). ACM.

Biggio, B., Corona, I., Fumera, G., Giacinto, G., & Roli, F. (2011). Bagging classifiers for fighting poisoning attacks in adversarial classification tasks. In *International workshop on multiple classifier systems* (pp. 350–359). Springer.

Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., et al. (2013). Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases* (pp. 387–402). Springer.

Biggio, B., Fumera, G., & Roli, F. (2014). Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering, 26*(4), 984–996.

Biggio, B., Nelson, B., & Laskov, P. (2012). Poisoning attacks against support vector machines. In *Proceedings of the 29th international conference on machine learning, ICML 2012, Edinburgh, Scotland, Uk, June 26 - July 1, 2012*.

Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., et al. (2014). Poisoning behavioral malware clustering. In *Proceedings of the 2014 workshop on artificial intelligent and security workshop* (pp. 27–36). ACM.

Budhraja, K. K., & Oates, T. (2015). Adversarial feature selection. In *2015 IEEE international conference on data mining workshop (ICDMW)* (pp. 288–294). IEEE.

Chakradeo, S., Reaves, B., Traynor, P., & Enck, W. (2013). Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on security and privacy in wireless and mobile networks* (pp. 13–24). ACM.

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on mobile systems, applications, and services* (pp. 239–252). ACM.

Chinavle, D., Kolari, P., Oates, T., & Finin, T. (2009). Ensembles in adversarial classification for spam. In *Proceedings of the 18th ACM conference on information and knowledge management* (pp. 2015–2018). ACM.

Dalvi, N., Domingos, P., Sanghai, S., & Verma, D. (2004). Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 99–108). ACM.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., et al. (2016). Droidscribe: Classifying android malware based on runtime behavior. In *Mobile security technologies (MoST 2016)* (pp. 1–12). 7kearns1993learning148

Deshotels, L., Notani, V., & Lakhotia, A. (2014). Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014* (p. 3). ACM.

Enck, W., Ongtang, M., & McDaniel, P. (2009). On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on computer and communications security* (pp. 235–245). ACM.

Feng, Y., Anand, S., Dillig, I., & Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering* (pp. 576–587). ACM.

Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., & Vigna, G. (2016). Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)* (pp. 377–396). doi:10.1109/SP.2016.30.

Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security, 5*(02), 56.

Garcia, J., Hammad, M., Pedrood, B., Bagheri-Khaligh, A., & Malek, S. (2015). Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Technical Report*. Department of Computer Science, George Mason University.

Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., & Rinard, M. C. (2015). Information flow analysis of android applications in droidsafe.. *NDSS*. Citeseer.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., & McDaniel, P. (2016). Adversarial perturbations against deep neural networks for malware classification. arXiv preprint arXiv:1606.04435

Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., & Tygar, J. (2011). Adversarial machine learning. In *Proceedings of the 4th ACM workshop on security and artificial intelligence* (pp. 43–58). ACM.

Kearns, M., & Li, M. (1993). Learning in the presence of malicious errors. *SIAM Journal on Computing, 22*(4), 807–837.

Klieber, W., Flynn, L., Bhosale, A., Jia, L., & Bauer, L. (2014). Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN international workshop on the state of the art in java program analysis* (pp. 1–6). ACM.

Labs, M. (2016). McAfee labs threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-may-2016.pdf. [Online; Accessed 19.07.2016].

Lakhotia, A., Walenstein, A., Miles, C., & Singh, A. (2013). Vilo: A rapid learning nearest-neighbor classifier for malware triage. *Journal of Computer Virology and Hacking Techniques, 9*(3), 109–123.

Laskov, P., & Lippmann, R. (2010). Machine learning in adversarial environments. *Machine Learning, 81*(2), 115–119. doi:10.1007/s10994-010-5207-6.

Laskov, P. (2014). Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy* (pp. 197–211). IEEE.

Lowd, D., & Meek, C. (2005). Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on knowledge discovery in data mining* (pp. 641–647). ACM.

Maiorca, D., Corona, I., & Giacinto, G. (2013). Looking at the bag is not enough to find the bomb: An evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on information, computer and communications security* (pp. 119–130). ACM.

Meng, G., Xue, Y., Mahinthan, C., Narayanan, A., Liu, Y., Zhang, J., et al. (2016). Mystique: Evolving android malware for auditing anti-malware tools. In *Proceedings of the 11th ACM on Asia conference on computer and communications security* (pp. 365–376). ACM.

Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., et al. (2013). Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX security symposium (USENIX security 13)* (pp. 543–558).

Pastrana, S., Orfila, A., & Ribagorda, A. (2011). A functional framework to evade network ids. In *System sciences (HICSS), 2011 44th Hawaii international conference on* (pp. 1–10). IEEE.

Perdisci, R., Gu, G., & Lee, W. (2006). Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *Sixth international conference on data mining (ICDM'06)* (pp. 488–498). IEEE.

Ptacek, T. H., & Newsham, T. N. (1998). Insertion, evasion, and denial of service: Eluding network intrusion detection. *Technical Report*. DTIC Document.

Rasthofer, S., Arzt, S., & Bodden, E. (2014). A machine-learning approach for classifying and categorizing android sources and sinks.. *NDSS*.

Russu, P., Demontis, A., Biggio, B., Fumera, G., & Roli, F. (2016). Secure kernel machines against evasion attacks. In *Proceedings of the 2016 ACM workshop on artificial intelligence and security* (pp. 59–69). ACM.

Sivanandam, S., & Deepa, S. (2007). *Introduction to genetic algorithms*. Springer Science & Business Media.

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

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27*(11), 1134–1142.

Valiant, L. G. (1985). Learning disjunction of conjunctions.. In *IJCAI* (pp. 560–566).

Vidas, T., & Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on information, computer and communications security* (pp. 447–458). ACM.

Vigna, G., Robertson, W., & Balzarotti, D. (2004). Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on computer and communications security* (pp. 21–30). ACM.

Wei, F., Roy, S., & Ou, X. (2014). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 1329–1341). ACM.

Xiao, H., Biggio, B., Brown, G., Fumera, G., Eckert, C., & Roli, F. (2015). Is feature selection secure against training data poisoning?. In F. Bach, & D. Blei (Eds.), *JMLR W&CP-proceedings of the 32nd international conference on international conference on machine learning (ICML): Vol. 37* (pp. 1689–1698).

Xu, W., Qi, Y., & Evans, D. (2016). Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*.

Xue, Y., Meng, G., Liu, Y., Tan, T. H., Chen, H., Sun, J., & Zhang, J. (2017). Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Transactions on Information Forensics and Security, 12*(7), 1529–1544.

Yang, C., Xu, Z., Gu, G., Yegneswaran, V., & Porras, P. (2014). Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *European symposium on research in computer security* (pp. 163–182). Springer.

Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., & Enck, W. (2015). Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE international conference on software engineering: Vol. 1* (pp. 303–313). IEEE.

Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 1105–1116). ACM.

Zheng, M., Lee, P. P., & Lui, J. C. (2012). Adam: an automatic and extensible platform to stress test android anti-virus systems. In *International conference on detection of intrusions and malware, and vulnerability assessment* (pp. 82–101). Springer.

Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy* (pp. 95–109). IEEE.

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS: Vol. 25* (pp. 50–52).