Quantifying the Impact of Adversarial Evasion Attacks on Machine Learning Based Android Malware Classifiers

Zainab Abaid*†, Mohamed Ali Kaafar‡† and Sanjay Jha*
*School of Computer Science and Engineering, University of New South Wales, Australia
{zainaba,sanjay}@cse.unsw.edu.au
‡Computing Department, Macquarie University, Australia
†CSIRO Data61, Australia
dali.kaafar@data61.csiro.au

***Abstract*—With the proliferation of Android-based devices, malicious apps have increasingly found their way to user devices. Many solutions for Android malware detection rely on machine learning; although effective, these are vulnerable to attacks from adversaries who wish to subvert these algorithms and allow malicious apps to evade detection. In this work, we present a statistical analysis of the impact of adversarial evasion attacks on various linear and non-linear classifiers, using a recently proposed Android malware classifier as a case study. We systematically explore the complete space of possible attacks varying in the adversary's knowledge about the classifier; our results show that it is possible to subvert linear classifiers (Support Vector Machines and Logistic Regression) by perturbing only a few features of malicious apps, with more knowledgeable adversaries degrading the classifier's detection rate from 100% to 0% and a completely blind adversary able to lower it to 12%. We show non-linear classifiers (Random Forest and Neural Network) to be more resilient to these attacks. We conclude our study with recommendations for designing classifiers to be more robust to the attacks presented in our work.**

I. Background and Introduction

With the rise in smartphones and other hand-held devices, mobile malware, particularly Android malware, has become a potent threat in recent years. Android malware detection has therefore received much attention from the research community [10]. Among the proposed detection approaches, machine learning remains a common thread, and has shown high accuracy [6], [15]. However, machine learning classifiers are threatened by *adversarial attacks* aiming to subvert the classifiers by poisoning their training or adapting malicious samples to evade detection [11]. The latter category, called *evasion* attacks, are the main focus of this paper. Evasion attacks are particularly important to consider in the context of mobile malware, as the push-based model allows developers to easily push updated versions of apps to users without any user involvement. Thus, malware authors can quickly respond to the detection of their apps by launching updated versions to evade detection. Thus, analysing the quantitative impact of adversarial evasion attacks on Android malware classifiers is a timely and important problem.

978-1-5386-1465-5/17/$31.00 ©2017 IEEE

Adversarial studies in the literature suffer a number of limitations: some make the assumption that an adversary has deep knowledge of the targeted classifier; others assume less knowledge but still assume some aspect of the classifier to be known [13]; still others are limited to specific problems and algorithms and their attack strategies do not hold for other domains [17]. Studies of adversarial attacks against mobile malware classifiers have only recently appeared in the literature [7]; however, a detailed analysis of the full range of possible evasion attacks against a variety of machine learning algorithms is not found in these efforts. In our work, we remove all assumptions of adversarial knowledge and systematically analyse the impact of a range of attacks, from fully informed to completely blind, on a typical Android malware classifier implemented with both linear and non-linear machine learning algorithms. To the best of our knowledge, this is the most comprehensive range of attacks studied in the Android malware detection domain to date.

To fully explore the space of evasion attacks on a classifier, we define a number of adversaries, differing in their knowledge of three key characteristics of a classifier (training data, feature set and classification algorithm), and formulate attack strategies for each adversary. To quantify the impact of these attack strategies on a typical Android malware classifier, as a case study we target DREBIN [6], which trains a Linear Support Vector Machine (SVM) and was shown to detect 94% of malware in a dataset of 123,453 apps. We find that DREBIN is susceptible to evasion attacks where the adversary can determine a set of features for each malicious app that can be perturbed to reduce its chances of being detected. Perturbing only a few features (<25) allows a fully informed adversary to lower DREBIN's detection rate from 100% to 0%, and a completely blind adversary to lower it to 12%. We also re-implement DREBIN with a number of different algorithms (Neural Network, Random Forest and Logistic Regression) and show that it can be made more resilient simply by replacing its Linear SVM with a non-linear classifier.

Overall, this paper makes the following contributions:

1) We propose adversarial strategies for modifying ma-

licious samples to evade detection against a range of adversarial capabilities;
2) We analyse the quantitative impact of evasion attacks on a real Android malware classifier, including a comparison with the resilience of other linear and non-linear classifiers to the same attacks;
3) We demonstrate a practical example of modifying a real malicious app to evade detection, showing that our proposed attacks represent a feasible threat.

We emphasise that while we have only demonstrated attacking a specific classifier, our attack strategies are general and not specifically adapted to the domain, and our study shows that such attacks represent a real threat to other learning-based malware classifiers.

The remainder of this paper is organised as follows. Section II details our threat model; Section III introduces DREBIN as our case study. Section IV outlines implementation details of our attacks. Section V presents our analysis of the impact of evasion attacks on DREBIN's linear SVM as well as other algorithms and a practical demonstration of an evasion attack. Section VI presents some recommendations for designing secure classifiers based on the insights from our study. Section VII outlines related work in this domain, and Section VIII concludes the paper.

II. Threat Model: Adversarial Goals, Capabilities and Strategies

We define a threat model from an adversary's perspective, comprising a goal and capabilities, and strategies for meeting the goal.

A. Goal

An Android malware classifier may face adversarial threats from malware authors whose apps may be detected by the classifier; a typical adversary would wish to update the apps such that they evade detection without compromising their malicious functionality. Thus, we set the adversarial goal of causing a classifier to mislabel a *target set* of malicious apps as benign.

B. Capabilities

An adversary's capability of carrying out a successful attack depends on its knowledge of three defining characteristics of the targeted classifier:

1) The labelled training data
2) The set of features to be extracted from the training data
3) The classification algorithm

In Table I, we enumerate a set of adversaries, each having a different level of capability depending on which of the three characteristics of a classifier are known, similar to the taxonomy defined in [13]. Each adversary has knowledge of a different subset of classifier components; Adversary **DFA**, for example, has access to the training **D**ata, the **F**eature set, and the classification **A**lgorithm, while Adversary **A** only knows the classification **A**lgorithm.

TABLE I: Different types of adversaries defined by their knowledge of the three key components of a classifier.

Adversary	Training Data (D)	Feature Set (F)	Classification Algorithm (A)
DFA	✓	✓	✓
DF	✓	✓	×
DA	✓	×	✓
D	✓	×	×
FA	×	✓	✓
F	×	✓	×
A	×	×	✓
NONE	×	×	×

C. Adversarial Strategies

The high level idea of the attack strategy we propose is outlined in Fig. 1. As the figure shows, each adversary takes the following steps to carry out an attack:

1) Step 1: Replicate the targeted classifier, which involves extracting a given set of features from training data to learn a model. The model assigns weights to features which represent the importance of each feature in causing a malicious classification.
2) Step 2: Extract the feature set from a *target* app which is to be modified to evade detection.
3) Step 3: Using the model learned in step 1, discover the *top features* of the app causing a malicious classification, as well as the top benign features in the model.
4) Step 4: Modify the app by removing its top malicious features and adding the top benign features to it.

As each adversary defined in Table I differs in terms of knowledge of the original classifier, the differences between the specific strategies of each adversary lie in how Step 1, i.e. replicating the classifier, is implemented. Adversary DFA, who has complete knowledge of the classifier, can simply replicate it without any additional steps. The adversaries that do not have access to the original training data, i.e. adversaries FA, F, A and NONE, additionally need to generate a *substitute* training set similar to the original data. As DREBIN's original data comprises apps downloaded from Android app markets, we build the substitute training using a different sample of apps from Google Play. Adversaries that do not have knowledge of the classification algorithm used by the targeted classifier, i.e. adversaries DF, D, F and NONE, additionally need to select *surrogate* algorithms that approximate the original classifier reasonably well. Both these strategies, i.e. using substitute training data and surrogate classification algorithms, have been successfully used in prior research [13]. However, to the best of our knowledge, no prior research has suggested a successful strategy for adversaries who do not have knowledge of the feature set used by the classifier, i.e. adversaries DA, D, A and NONE. Trivial obfuscation techniques have been attempted but shown not to work [9]. We propose a "feature-guessing" strategy, where the adversary relies on domain

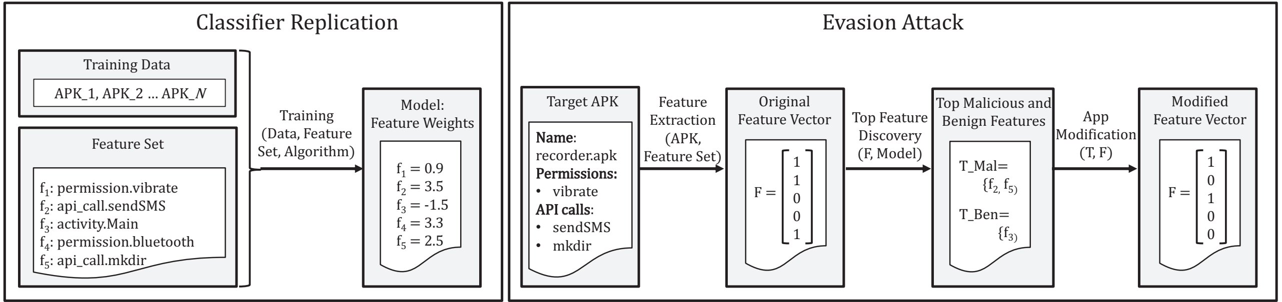


Fig. 1: Process followed by an adversary to carry out evasion attacks.

knowledge to approximate the feature set. In the Android domain, many earlier proposed solutions for malware detection have relied on permissions, for example [24], and a recent solution successfully uses the API call graph of apps to classify them [15]. The adversary can use this knowledge to construct a feature set comprising API calls and permissions. Once an adversary has selected an original or a substitute training set, extracted the original or a substitute set of features from it, and run the original or a surrogate classification algorithm, the classifier replication step can be carried out to obtain a model, after which Steps 2 – 4 (Evasion Attack) can be performed.

III. An Introduction to DREBIN

In this section we introduce the targeted classifier that we use as a case study for testing the effectiveness of our proposed adversarial strategies. DREBIN [6], proposed in 2014, detects Android malware by deploying a trained Linear SVM on smartphones. We chose DREBIN to demonstrate the impact of adversarial attacks against machine learning based mobile malware classifiers for a number of reasons. First, DREBIN is clearly vulnerable to adversarial evasion attacks, as it is possible for an adversary to leverage knowledge of DREBIN to modify malicious samples to evade detection. Secondly, DREBIN is a recent approach and represents the state of the art in identifying Android malware on mobile devices; a study investigating the feasibility of attacks against such a classifier is therefore much-needed. Finally, our focus on Android malware, as opposed to general threats, is motivated by the open nature of the Android app market, which makes it relatively easy for attackers (for example, malware authors) to modify apps and publish new variants to avoid detection. We now describe this classifier in terms of the three defining characteristics, i.e. the feature set, the classification algorithm, and the data that was used in its training.

A. Feature Set

DREBIN extracts features from the source code and manifest file of an app from the following categories.

1) Permissions from the manifest file.
2) API calls from the source code of the app.
3) App components (activities, services, receivers, providers) from the manifest file.
4) Filtered Intents from the manifest file.
5) Hardware features (hardware components used by an app) from the manifest file.
6) Network Addresses (IP/URL) from the source code.

The full set $[S]$ of features from the training data comprises thousands of elements (545000 in the original DREBIN dataset). The features extracted from each app are mapped to an $|S|$-dimensional vector where each dimension is 1 or 0 depending on whether or not the app has the feature represented by that dimension. Fig. 1 shows an example where the original feature set has 5 features; thus, the app's features are mapped to a 5-dimensional feature vector where the two dimensions representing its features are set to 1.

B. Classification Algorithm

A linear SVM is trained over the extracted features from the data to learn a vector $w \in \mathbb{R}^{|S|}$ specifying the direction of a separating hyperplane. Each element of w can be interpreted as a weight assigned to the corresponding feature. The classification of each app in a linear SVM where the features are all binary is simply found by calculating $f(x) = \sum w_s$, where w_s is the weight of each feature. If $f(x) < 0$, the classification is benign, otherwise malicious. Thus, the top features causing a malicious classification can be determined simply by noting the features with the largest k positive weights.

C. Training Data

The training dataset comprises 96,150 apps from Google Play, 22,355 apps from different alternative Chinese or Russian Markets, and 13,106 samples from blogs and forums. Each app was labelled by uploading it to VirusTotal [5]; apps declared malicious by two or more of ten well known anti-viruses (AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos) were labeled malicious. 123,453 benign and 5,560 malicious apps comprise the final dataset.

IV. Prerequisite Implementation Tasks

In this section we describe some pre-requisite implementation tasks required for quantifying the impact of attacks on DREBIN.

A. Replicating DREBIN

DREBIN is not publicly available, so we obtained the training data released by its authors and implemented feature extraction logic based on the details provided in the paper [6]. We used `apktool` to decompile the apps in the data and wrote Java code to parse the xml-format manifest file and the Smali code to extract the features. We used LibLinear to train a Linear SVM. Despite some differences in the calculated features (e.g. missing names of activities or extra API calls added), our DREBIN clone was able to approximate the accuracy of the original DREBIN, with it true positive rate (TPR) of 92% closely matching the original TPR of 94% reported in the paper.

B. Constructing a Substitute Training Dataset

We collected 7000 apps from Google Play, using a crawler implemented by the authors of [12] and used the VirusTotal API to filter out apps detected as malicious by two or more of VirusTotal's scanners, which left us with 3,862 benign apps. For constructing the malicious part of the dataset, we used 1,788 malicious apps from DREBIN's dataset (dropping them from the training set of our version of DREBIN). Our final substitute training dataset comprised 5,650 apps[1].

C. Selecting a Substitute Feature Set

A common thread in prior research in Android malware detection is the use of permissions declared in the Android manifest [24]. One recent work has also used the set of API calls in an app to build an API call graph to detect malicious apps [15]. Jointly, the API calls and the permissions of an app should provide reasonably good discrimination between malicious and benign samples; we use this as our substitute feature set. In practice, an adversary may be forced to choose features completely orthogonal to those used by the original classifier, but because DREBIN already extracts as many features as possible from an app, we were unable to select a completely orthogonal feature set. However, this is a consequence of the specific classifier we target and not a methodological problem.

D. Selecting Surrogate Classifiers

We chose a number of well known machine learning algorithms to run over the Drebin dataset, including linear classifiers (Logistic Regression, Percepton, Stochastic Gradient Descent, Passive-Aggressive Classifier), ensemble learning methods (Random Forest, AdaBoost), Neural Network and Decision Tree. We noted the TPR as well as the overall accuracy (percentage of correctly labeled samples) of these algorithms on the original Drebin dataset, the purpose being to select the algorithm that achieved an accuracy closest to that of Drebin's Linear SVM. Fig. 2 shows the results. Most classifiers performed quite badly in terms of TPR, with linear classifiers performing better than non-linear ones[2]. Logistic Regression achieved the closest TPR and accuracy to Drebin's Linear SVM, (99.1% accuracy compared to Drebin's 99.2%), followed by Neural Network, Random Forest, AdaBoost, the remaining linear classifiers and finally Decision Tree which only achieved 51% accuracy. Thus, we chose Logistic Regression as a surrogate classifier for our attacks on DREBIN. Logistic Regression is a linear method similar to a Linear SVM and thus likely to approximate it closely. In a separate experiment, discussed in Section V-C, we investigate the effect of using non-linear algorithms as surrogates, using Random Forest (95.7% accuracy) as an example, as it is a non-linear ensemble learning method. We used scikit-learn [4] for implementing Random Forest and LibLinear for Logistic regression.

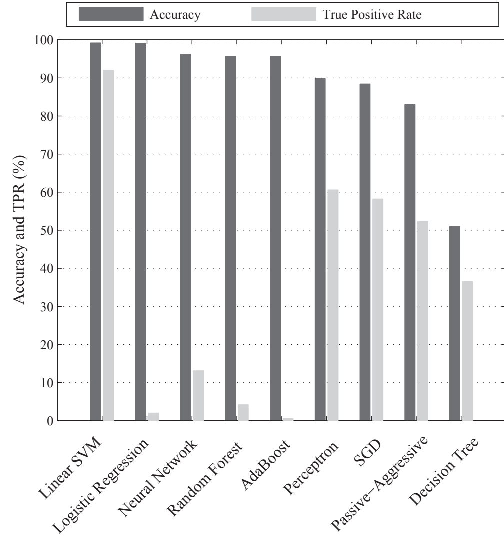


Fig. 2: Detection accuracy and true positive rate (TPR) achieved by different classifiers on the Drebin dataset.

E. Constructing a Target Set and Discovering Top Features

The target set of malicious apps should (a) not be in any classifier's training set, and (b) be classified malicious by DREBIN. We split the original DREBIN data into training and testing data in a 70/30 ratio. The training data was used to train the replicated classifier, and the malicious apps from the testing data that were detected by DREBIN became our target set, comprising 1,010 malicious apps. Finally, we wrote Java code to find the top features of each app in the target set, simply by sorting each feature of the app by its weight according to the model used by a given adversary.

[1]Varying the construction of the substitute dataset, for example by using malicious apps from public repositories [1], or only using apps from the same time frame as those in DREBIN's original training dataset, was left as future work.

[2]This may be because the original DREBIN dataset is unbalanced; malicious samples form less than 5% of the data. Using a dataset with an equal number of benign and malicious samples improved the TPR (e.g. Random Forest went from 4% to 50%).

TABLE II: Removing features from each of DREBIN's feature categories.

Feature Category	Removable?	Reason
Permissions	No	The Android permission system requires declaring permissions in the manifest. Permissions not requested but actually used can be detected by DREBIN.
API Calls	Yes	API method names can be encoded, and decoded and called dynamically using reflection.
App Components	Yes	Activities, services, receivers and providers are implemented as Java classes and listed in the manifest file, so they can simply be renamed.
Filtered Intents	No	Intents are listed in the manifest file and have standard names which cannot be renamed or encoded to look different.
Hardware Features	No	The Android permission system requires declaring hardware usage, e.g. `uses-microphone` in the manifest file.
Network Address	Yes	They can be encoded and only decoded at run-time so as to be unrecongnisable during the static analysis.

V. Evaluation: Quantifying the Impact of Attacks on DREBIN

In this section, we quantify the impact of adversarial attacks on a variety of algorithms. As discussed in Section II-C, an evasion attack involves removing the most incriminating features of malicious apps as well as adding some benign features to them. A real adversary will make these modifications in the source code of an app, for example by adding extra functions or encoding object names. However, performing these modifications in practice requires significant manual effort and is difficult given the large number of apps in our target set, especially as the source code of the apps has to be obtained using decompilation tools and is often obfuscated or full of errors. Thus, we performed the attacks entirely in feature space by directly modifying feature vectors rather than changing the source code (similar to [9], [20]). Specifically, as the features of apps are stored in a binary representation in feature vectors (as shown in the example in Fig. 1), we hid incriminating features simply by setting their values to 0 instead of 1 in the feature vector, and likewise added benign features by setting their values to 1 if they were 0 originally. We present, at the end of this section, a demonstration of how we modified one real malicious app to evade detection by disguising some features in the source code. Overall however, this work has a theoretical focus and represents a statistical analysis of what the impact of these attacks would be in practice.

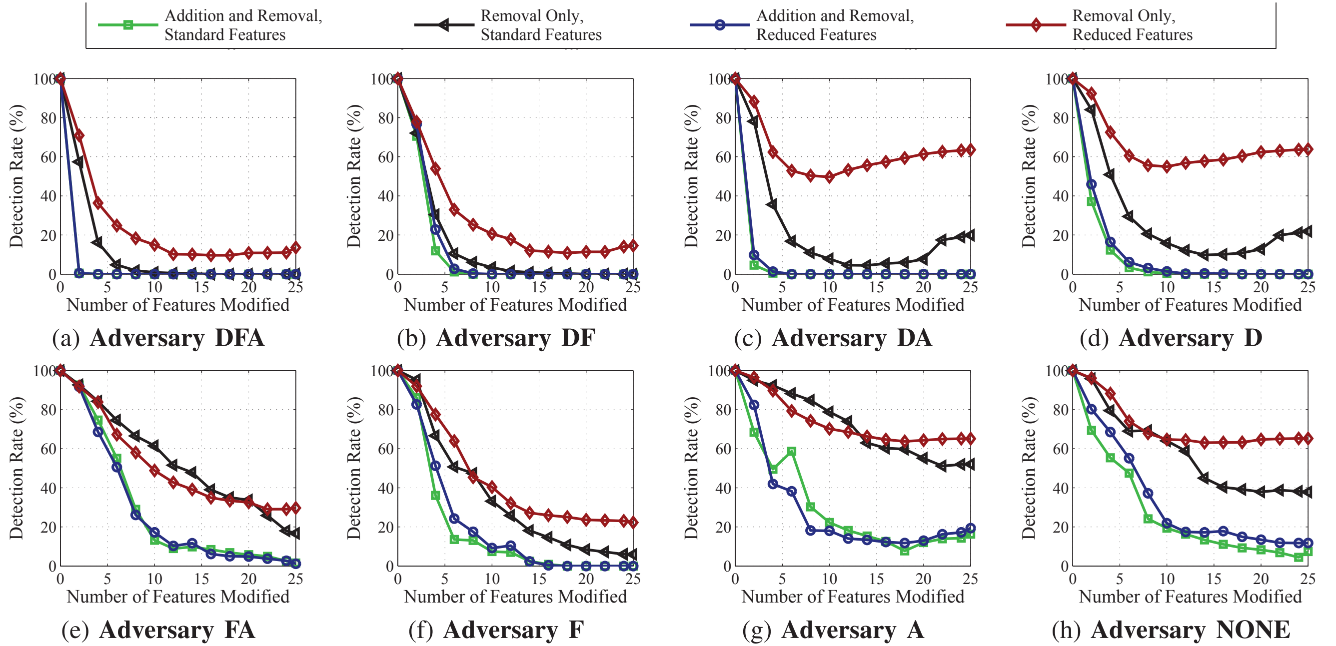
Practical Considerations: In practice, adding benign features to an app can be achieved by adding new methods or classes that will not disturb the original app functionality. Hiding certain malicious features such that the app's malicious functionality is not compromised, however, may be challenging. Table II shows that of the different categories of features extracted by DREBIN, API calls, app components, and network addresses can all be renamed or encoded to look different but retain the same functionality, evading at least static analysis based approaches. However, *permissions*, *hardware features* and *intents* cannot simply be dropped, as they usually have predefined names and must be declared in the manifest file for an app to run successfully. Thus, we define different experimental settings for our attacks as follows.

Experimental Settings: We refer to the full set of features as *Standard Features* and the set of features from the three modifiable categories according to Table II, i.e. API calls, app components and network addresses as *Reduced Features*, and consider four different adversarial settings. In *Addition and Removal, Standard Features*, the adversary can freely add or remove any feature of an app. In *Addition and Removal, Reduced Features*, the adversary can add any feature but remove only features belonging to the reduced set. In *Removal Only, Standard Features*, the adversary cannot add features but may remove any feature, and in *Removal Only, Reduced Features*, the adversary can only remove features from the reduced set. The results observed in settings where any feature can be removed represent the theoretical limit of an attack's effectiveness, and can only be achieved in practice if the adversary can disguise the permission, hardware and intent features without compromising the app. Results for the remaining settings are practically achievable against static analysis based approaches. Practical evasion of classifiers that extract more complex and dynamic features may be a more difficult problem and one that we leave for future research.

A. Relationship Between Adversarial Knowledge and Attack Effectiveness

Our analysis is shown in Fig. 3, where each plot represents an attack by an adversary having knowledge of a different subset of the three classifier components. *Removing* n features from an app means that the adversary has changed n elements of the feature vector from 1 to 0, while *adding and removing* n features means that the adversary has changed n elements from 1 to 0 and another n elements from 0 to 1. We now discuss the effect of knowing each classifier component on the effectiveness of an attack.

Training data: Adversaries in this category are able to access the original training data of the targeted classifier. Figures 3(a) to 3(d) show these attacks to be highly effective; Drebin's detection rate on the target set is reduced to 0% in the *Addition and Removal, Reduced Features* setting. However, attacks in the *Removal Only* categories are less effective;



(a) **Adversary DFA** (b) **Adversary DF** (c) **Adversary DA** (d) **Adversary D**

(e) **Adversary FA** (f) **Adversary F** (g) **Adversary A** (h) **Adversary NONE**

Fig. 3: Degradation in Drebin's detection rate as a result of attacks by adversaries with varying levels of knowledge about the classifier. Where the algorithm is unknown (DF, D, F and NONE) the surrogate algorithm is Logistic Regression.

Adversary D is only able to reduce Drebin's detection rate to 55% when features can only be removed from apps, compared to 0% when features can be added as well. In some attacks, Drebin's detection rate rises again as more features are modified; this usually occurs when an app only has a few positively weighted features – after they get selected for removal, features with negative weights (associated with benign classification) begin to get removed, increasing chances of a malicious classification. This point therefore represents the limit to which Drebin's detection rate can be degraded.

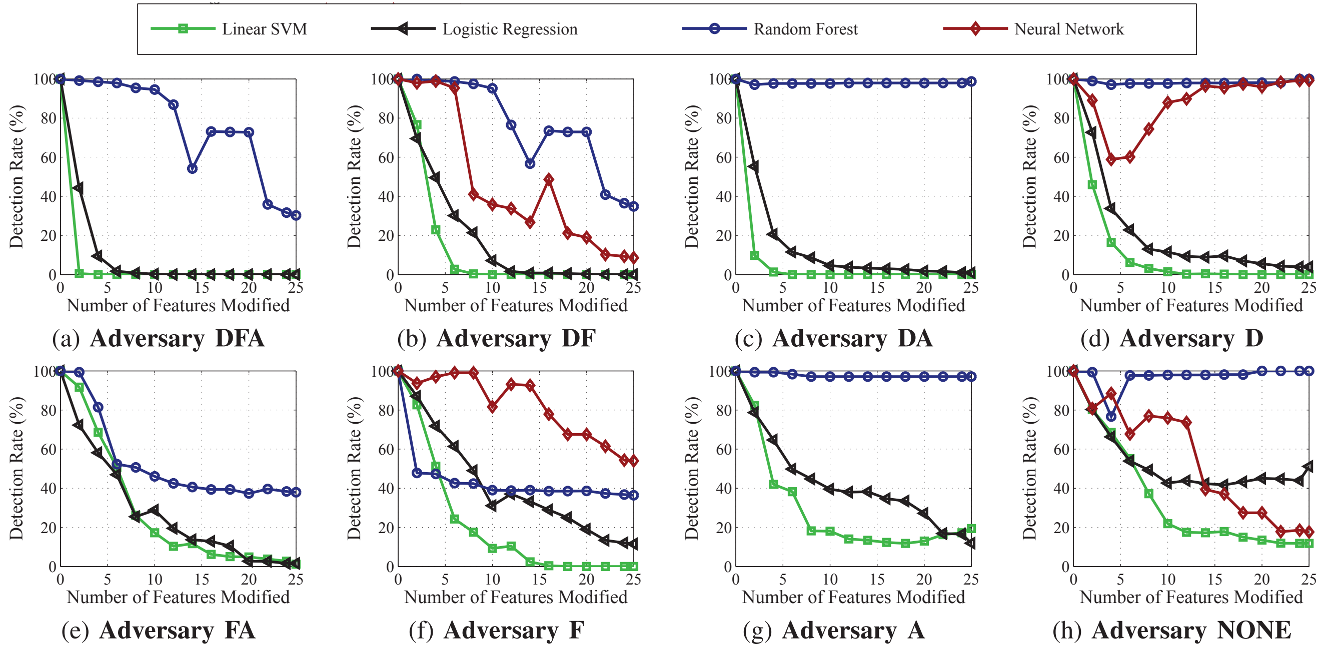
Feature Set: Adversaries FA and F are aware of the feature set used by the classifier but do not have access to its original training data. As shown in Fig 3(e) and (f), despite using substitute data to train a replicate model of Drebin, both adversaries can degrade Drebin's detection rate to 0% if they can add as well as remove features, even from a reduced feature set. When only able to remove features, both still degrade the detection rate to below 30%.

Algorithm: We determine the importance of knowing the algorithm by comparing the performance of adversaries who differ only in knowledge of the algorithm. Based on a comparison of Adversary DFA's attack with that of Adversary DF, of Adversary DA's attack with that of Adversary D, and so on, we conclude that knowledge of the algorithm makes little difference, and adversaries unaware of it can generate attacks equally effective as those aware of it. In fact, in case of a completely blind attack (Adversary NONE), a surrogate algorithm may even perform better than the original. However, it is important that the adversary uses a surrogate algorithm to train a replicate model of the targeted classifier that closely approximates the original. In the results reported in this figure, the adversaries use Logistic Regression as a surrogate, which is also a linear classification algorithm and similar to the Linear SVM. We explore the effect of using different surrogate algorithms in Section V-C.

Overall, the results show even the most limited adversaries (Adversary A and Adversary NONE) can reduce DREBIN's detection rate to 12% from an intial 100%, and those with knowledge of the feature set or training data alone can reduce it to 0%. DREBIN implemented as a Linear SVM therefore appears to be highly vulnerable to evasion attacks. We acknowledge that these results cannot necessarily be generalised to all machine learning classifiers, as our analysis is currently limited to the Android malware domain and static feature extraction. Investigating whether similar results can be achieved against learners in different settings is left as an open problem.

B. Resilience of Linear Vs. Non-Linear Classification Algorithms to Attacks

Our attacks on DREBIN have highlighted the vulnerability of the Linear SVM to adversarial attacks based on feature perturbation. We now evaluate the impact of the same attacks against another linear classifier (Logistic Regression), a non-linear classifier (Random Forest), and another non-linear classifier (Neural Network, as it has recently gained much popularity in deep-learning applications). We do this by implementing modified versions of DREBIN that run one of these algorithms instead of a Linear SVM, and implement the attacks exactly as with DREBIN, for all eight adversaries. We only show results



(a) **Adversary DFA** (b) **Adversary DF** (c) **Adversary DA** (d) **Adversary D**

(e) **Adversary FA** (f) **Adversary F** (g) **Adversary A** (h) **Adversary NONE**

Fig. 4: Comparison of the resilience of various algorithms to attacks by adversaries with varying levels of knowledge. Best performing surrogate algorithm is used for attacking each algorithm by adversaries unaware of the targeted classifier's algorithm (i.e. DF, D, F and NONE).

for the *Addition and Removal, Reduced Features* setting as it represents a practically feasible setting. We also assume that adversaries that are unaware of the targeted algorithm (Adversaries DF, D, F and NONE) choose the following (best) surrogates: Logistic Regression for attacking Linear SVM and Neural Network, Linear SVM for attacking Logistic Regression, and AdaBoost for attacking Random Forest. Section V-C presents a comparison of the effectiveness of different surrogate algorithms in attacking a given classifier. Finally, the Neural Network has not been attacked by adversaries who are aware of the algorithm (Adversaries DFA, DA, FA and A) as it does not generate a model that can be intuitively understood or mapped to feature weights. Adversaries cannot determine the most important features to be modified by replicating the Neural Network, but will instead have to use a surrogate algorithm to determine important features; therefore, knowledge of the algorithm brings no advantage, and, for example, an attack by Adversary DFA will be the same as that by Adversary DF. Thus, we do not show attacks on the Neural Network by these adversaries.

Fig. 4 shows that both linear classifiers – Logistic Regression and Linear SVM – are significantly more vulnerable to adversarial attacks in most scenarios compared to the non-linear classifiers, with Logistic Regression showing slightly more resilience than the Linear SVM as its detection rate degrades more gradually, especially when the adversary has limited knowledge (adversaries F, A and NONE). In attacks against the Neural Network, knowledge of the feature set appears important when training data is known: Adversary DF performs a more effective attack than Adversary D. However, when the adversary uses a substitute training set, a blind attack (Adversary NONE) performs better than an attack in which the feature set is known (Adversary F). Adversary NONE is unaware of the feature set and can modify a very limited set of features; it appears that this yields a better attack for the substitute training data. This is not predictable behaviour and a real adversary may need to try perturbing various feature subsets to see which yields the best attack. The Random Forest classifier proves to be the most difficult to successfully attack, showing almost no degradation when the feature set is unknown to the adversary (Adversay DA, D, A and NONE). Adversaries having access to training data (DFA, DF) outperform those without it (FA, F), but knowing the algorithm makes little difference in attacking this classifier.

Based on this analysis, linear classifiers appear clearly more susceptible to attack than non-linear ones for our experiments. While in other settings, non-linear classifiers may not necessarily be more resilient, the insight gained from our experiments is that when designing a malware classifier, it may be advantageous to implement different versions of it with different algorithms, simulate adversarial attacks against all versions, and select the most resilient version. In this case, replacing DREBIN's Linear SVM with the Random Forest algorithm greatly improves its resilience to attacks.

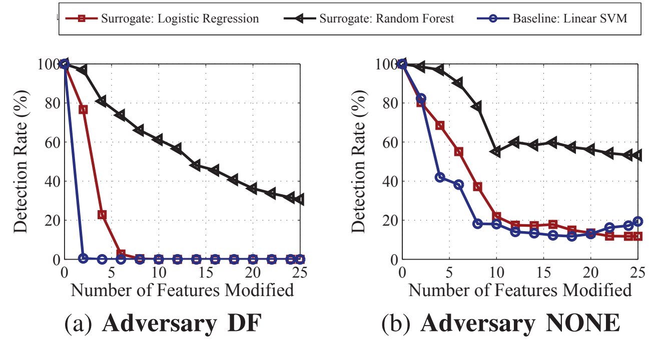
C. Comparison of Surrogate Algorithms

We now analyse the effect of using different surrogate algorithms for attacking both linear and non-linear classifiers with no knowledge of the targeted classification algorithm.

The purpose is to analyse whether linear algorithms always approximate other linear algorithms better than non-linear classifiers do, and vice versa. Fig 5 shows the degradation in the detection rate of Drebin's Linear SVM as Adversary DF and Adversary NONE attack it using Logistic Regression and Random Forest as surrogate algorithms. Results for Adversary D show a trend similar to Adversary DF, and those for Adversary F are similar to Adversary NONE; we omit these for brevity. Fig 6 shows the corresponding results for attacking Drebin implemented as a Logistic Regression classifier, using Logistic Regression and Random Forest as surrogates. We also show, as a baseline for comparison, the degradation achieved when the adversary has knowledge of the algorithm – e.g. attacking a Linear SVM using another replicated Linear SVM. We observe that attacks against Linear SVM and Logistic Regression are most effective when a linear classifier is used as a surrogate (i.e. Logistic Regression for Linear SVM and vice versa), while using Random Forest as the surrogate yields a significantly less effective attack.

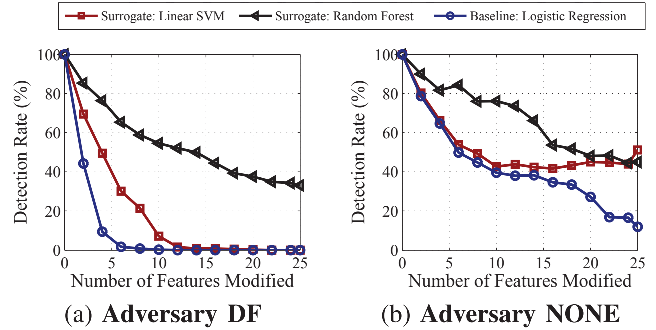
Fig 7 shows results for attacking Random Forest using Logistic Regression and AdaBoost as surrogates. In Adversary DF's attack, Logistic Regression, predictably, does not approximate the Random Forest classifier well, and using it as a surrogate yields the least effective attack. AdaBoost, however, yields an attack as effective as it would be if the algorithm was known, i.e. if Random Forest itself was used to build a replicate model. In the blind attack by Adversary NONE, however, using Random Forest itself is almost completely unsuccessful in degrading the detection rate, while AdaBoost initially performs better and achieves the minimum rate of 78%. As more features are modified (> 4), Logistic Regression achieves the maximum degradation (82%) while AdaBoost and Random Forest are completely unsuccessful. If Linear SVM is used instead of Logistic Regression (not shown in the figure), the results remain very similar. Thus, both linear classifiers perform the same. Overall, as the key metric we consider is the minimum detection rate achieved, we conclude that AdaBoost forms a better surrogate for attacking Random Forest rather than a linear classifier.

Fig 8 shows results for attacking a Neural Network. We do not show a baseline as a Neural Network cannot be attacked by building a replicate Neural Network model. We attack the Neural Network using Linear SVM, Logistic Regression and Random Forest as surrogates. For the adversary with more knowledge (Adversary DF) Linear SVM is the best surrogate, but is closely followed by both Logistic Regression and Random Forest. For a blind adversary (NONE), using Linear SVM does not degrade the detection rate beyond 23%, after which the detection rate rises again. In scenarios with less knowledge and fewer features available to modify, Random Forest and Logistic Regression perform almost equally well. Thus, Figures 5 to 8 show that linear algorithms certainly form better surrogates for other linear algorithms to be attacked, but non-linear algorithms *may or may not* form better surrogates for non-linear algorithms; different levels of adversarial knowledge may show different trends.



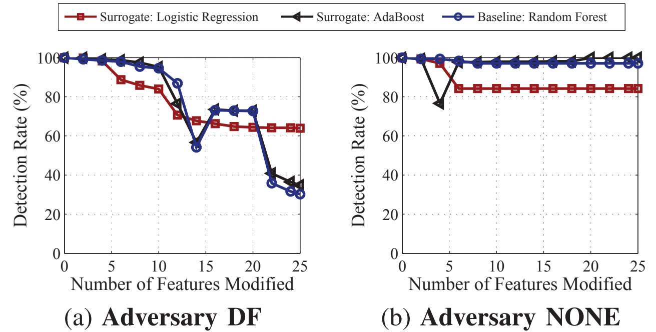
(a) **Adversary DF** (b) **Adversary NONE**

Fig. 5: Attacking Linear SVM.



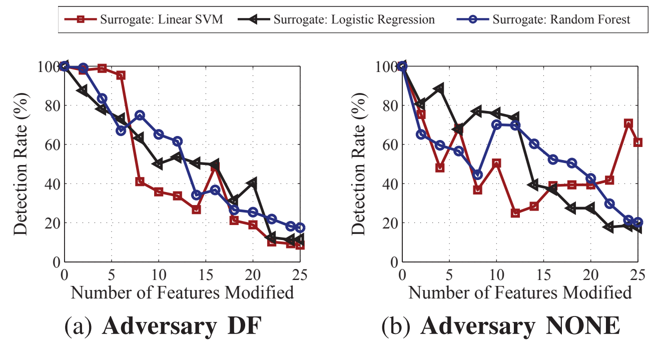
(a) **Adversary DF** (b) **Adversary NONE**

Fig. 6: Attacking Logistic Regression.



(a) **Adversary DF** (b) **Adversary NONE**

Fig. 7: Attacking Random Forest.



(a) **Adversary DF** (b) **Adversary NONE**

Fig. 8: Attacking Neural Network.

TABLE III: Top-weighted 15 features of a malware sample from the DroidKungFu family.

Category	Feature List
API Calls	`android/net/wifi/WifiManager /getWifiState, java/net/URL/openConnection, android/net/ConnectivityManager /getNetworkInfo, java/net/HttpURLConnection, android/telephony/TelephonyManager /getDeviceId, android/net/wifi /WifiManager/setWifiEnabled, android/telephony/TelephonyManager /getLine1Number`
Restricted API Calls	`system/bin/su, Cipher(ad), HttpPost, printStackTrace, getWifiState, getSystemService`
Components	`com.google.search.Receiver`

D. Practical Demonstration of Evasion Attack

We now demonstrate a practical evasion attack by modifying the source code of a malicious app detected by DREBIN. We obtained a variant of the DroidKungFu malware and its source code from Project Kharon [3] which is detected by our DREBIN clone. We then extracted the top-weighted 25 features, excluding permissions, hardware features or intents, as they are not easy to hide. By removing one feature at a time, we found that removing 15 of these 25 features, summarised in Table III, allowed the app to evade detection. To hide API calls, we implemented a simple encoding scheme that replaces each character of a string with the next character in the ASCII table, and replaced the original API calls with their encoded names. We wrote a decoding function to obtain the original API calls at runtime, and then used reflection to dynamically invoke the decoded calls. Thus, the functionality of the call was retained but it could not be picked up by DREBIN. While this strategy worked for evading static analysis based approaches, classifiers that perform dynamic analysis would still see the original API calls once they are invoked. We left this as an open issue for future work. The app component feature (`com.google.search.Receiver`) is a receiver, represented as a Java class; we hid this by renaming the Java class. Our adversarial strategy was therefore a form of obfuscating the features such that DREBIN's feature extraction module would be unable to find them, as actually removing features would compromise the malicious functionality of the app. We then recompiled the modified app and ran DREBIN to obtain the modified feature vector, and verified that the new feature vector no longer contained the top 15 features. Classifying the modified feature vector using our version of DREBIN yielded a benign classification, showing that the attacks we implement in feature space are viable in reality and present a valid threat to mobile malware classifiers.

VI. RECOMMENDATIONS

Based on the insights gained in this study, we make some recommendations for designing machine learning based Android malware classifiers in order to achieve robustness against adversarial attacks:

1) As knowledge of the feature set appears to play a very important role in the effectiveness of an attack, frequently retraining a classifier using different random subsets of the full feature set may make it difficult for adversaries to replicate the classifier, as they do not know which features the current version of the classifier is extracting.
2) Building classifiers to use features that are difficult to remove without affecting the malicious functionality of the app would allow less room for an adversary to modify samples to evade detection, and defeat an adversarial strategy based on feature perturbation.
3) Classifier-fusion, under which several algorithms are run on different training sets and the results of a random subset of the algorithms is combined to generate a classification result [22], can make the feature ranking (i.e. calculation of feature weights) unpredictable, lowering the success of adversarial strategies based on removing highly weighted features.

VII. RELATED WORK

The vulnerability of machine-learning based approaches to adversarial attacks has been frequently acknowledged in the literature, and several efforts have been made to taxonomise these attacks and discuss countermeasures [8]. Training poisoning attacks, which involve inserting deliberately misleading data into a classifier's training set, have been demonstrated against, among others, spam filters [16], and network anomaly detectors [18]. Evasion attacks such as those presented in our work have targeted anomaly-based intrusion detection systems [21], as well as spam filters [14]. The limitation of most of these attacks is assuming a highly knowledgeable adversary. As this may not be true in practice, we remove all such assumptions in our work.

Recently, more realistic attack approaches have been proposed that assume less adversarial knowledge [23]. A close inspiration for our research is [13], where the authors propose a range of attacks against a malicious PDF file detector, assuming limited adversarial knowledge. However, knowledge of the feature set used by the classifier is still assumed. Some completely blind attacks have been proposed very recently [17] but evaluated only on deep learning neural networks for image classification, in which the feature set used by the classifier is implicitly known, as it is simply the set of pixel values of an image. Thus, the problem of not knowing the feature set has not been addressed. Completely blind attacks have also been attempted, albeit unsuccessfully, in a very recent effort that is closely related to our work [9] and also simulates attacks against DREBIN. However, unlike our work, not all eight possible variations of adversarial knowledge have been

investigated, and the resilience of linear and non-linear algorithms has not been compared, nor the effect of using different surrogate algorithms to approximate classifiers. In fact, as the focus of the work in [9] is on proposing security measures against such attacks, its main contribution remains orthogonal to ours. Investigating defences against our proposed attacks is currently beyond the scope of our work; thus, other efforts towards proposing solutions against adversarial attacks [19] remain orthogonal to ours.

Overall, to the best of our knowledge, ours is the first work in which we (a) quantify the impact of the full range of evasion attacks given all possible levels of adversarial knowledge, and (b) compare the resilience of linear and non-linear classifiers to adversarial attacks.

VIII. Conclusion and Future Directions

In this work we have studied the vulnerability of machine learning based Android malware classifiers to adversarial evasion attacks aiming to cause misclassification of malicious mobile apps. As a case study, we have performed diverse evasion attacks, ranging from fully informed to completely blind attacks, against DREBIN, an Android malware classifier implemented as a Linear SVM and other linear and non-linear classifiers. We show that adversaries with perfect knowledge of a linear classifier can degrade its detection rate from 100% to 0%, and even completely blind adversaries can lower it to 12%. However, non-linear classifiers are more resilient to such attacks. We practically demonstrate an evasion attack for a real malicious app and show that the attacks we have investigated are practically possible and represent a valid threat to mobile malware classifiers; moreover we make recommendations for designing robust classifiers based on the insights gained from our study. We acknowledge that we have only demonstrated attacking one classifier in certain settings; it remains to be investigated whether our results can be generalised to all machine learning classifiers for Android or other malware. It is possible that classifiers in other settings may show different levels of vulnerability to adversarial attacks. An important future direction in this regard is to investigate whether using more complex, dynamically extracted features as opposed to the statically extracted simple features used by DREBIN increases a classifier's robustness against attacks. Secondly, practically testing the security recommendations we have made for designing robust classifiers, especially the idea of classifier fusion, remains an open problem. Overall, we believe that our work not only contributes significantly to current adversarial research by carrying out a comprehensive analysis of the impact of evasion attacks, but also opens many interesting avenues for future research in this area.

References

[1] Androzoo. https://androzoo.uni.lu/, 2017. [Online; accessed 12-Sep-2017].

[2] International Data Corporation Worldwide Quarterly Mobile Phone Tracker. http://www.idc.com/promo/smartphone-market-share/os, 2017. [Online; accessed 12-Sep-2017].

[3] Project Kharon: Studying Android Malware Behaviours. http://kharon.gforge.inria.fr/, 2017. [Online; accessed 12-Sep-2017].

[4] Scikit Learn: Machine Learning in Python. http://scikit-learn.org, 2017. [Online; accessed 12-Sep-2017].

[5] VirusTotal. https://virustotal.com/, 2017. [Online; accessed 12-Sep-2017].

[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of Android malware in your pocket. In *NDSS*, 2014.

[7] S. Chen, M. Xue, and L. Xu. Towards adversarial detection of mobile malware: poster. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 415–416. ACM, 2016.

[8] I. Corona, G. Giacinto, and F. Roli. Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues. *Information Sciences*, 239:201–225, 2013.

[9] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[10] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[11] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[12] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 349–364. ACM, 2016.

[13] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[14] D. Lowd and C. Meek. Good word attacks on statistical spam filters. In *CEAS*, 2005.

[15] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *NDSS*, 2017.

[16] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. Rubinstein, U. Saini, C. Sutton, J. Tygar, and K. Xia. Misleading learners: Co-opting your spam filter. In *Machine learning in cyber trust*, pages 17–51. Springer, 2009.

[17] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[18] B. I. Rubinstein, B. Nelson, L. Huang, A. D. Joseph, S.-h. Lau, S. Rao, N. Taft, and J. Tygar. Antidote: understanding and defending against poisoning of anomaly detectors. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 1–14. ACM, 2009.

[19] P. Russu, A. Demontis, B. Biggio, G. Fumera, and F. Roli. Secure kernel machines against evasion attacks. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, pages 59–69. ACM, 2016.

[20] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248. ACM, 2012.

[21] K. M. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *International Workshop on Recent Advances in Intrusion Detection*, pages 54–73. Springer, 2002.

[22] K. Woods, W. P. Kegelmeyer, and K. Bowyer. Combination of multiple classifiers using local accuracy estimates. *IEEE transactions on pattern analysis and machine intelligence*, 19(4):405–410, 1997.

[23] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli. Is feature selection secure against training data poisoning? In *ICML*, pages 1689–1698, 2015.

[24] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *NDSS*, volume 25, pages 50–52, 2012.