Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection

Ambra Demontis, *Student Member, IEEE*, Marco Melis, *Student Member, IEEE*, Battista Biggio, *Senior Member, IEEE*, Davide Maiorca, *Member, IEEE*, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, *Senior Member, IEEE*, and Fabio Roli, *Fellow, IEEE*

Abstract—To cope with the increasing variability and sophistication of modern attacks, machine learning has been widely adopted as a statistically-sound tool for malware detection. However, its security against well-crafted attacks has not only been recently questioned, but it has been shown that machine learning exhibits inherent vulnerabilities that can be exploited to evade detection at test time. In other words, machine learning itself can be the weakest link in a security system. In this paper, we rely upon a previously-proposed attack framework to categorize potential attack scenarios against learning-based malware detection tools, by modeling attackers with different skills and capabilities. We then define and implement a set of corresponding evasion attacks to thoroughly assess the security of Drebin, an Android malware detector. The main contribution of this work is the proposal of a simple and scalable secure-learning paradigm that mitigates the impact of evasion attacks, while only slightly worsening the detection rate in the absence of attack. We finally argue that our secure-learning approach can also be readily applied to other malware detection tasks.

Index Terms—Android malware detection, static analysis, secure machine learning, computer security

1 Introduction

During the last decade, machine learning has been increasingly applied in security-related tasks, in response to the increasing variability and sophistication of modern attacks [1], [3], [6], [27], [33]. One relevant feature of machine-learning approaches is their ability to *generalize*, i.e., to potentially detect never-before-seen attacks, or variants of known ones. However, as first pointed out by Barreno et al. [4], [5], machine-learning algorithms have been designed under the assumption that training and test data follow the same underlying probability distribution, which makes them vulnerable to well-crafted attacks violating this assumption. This means that machine learning itself can be the weakest link in the security chain [2]. Subsequent work has confirmed this intuition, showing that machine-learning techniques can be significantly affected by carefully-crafted attacks exploiting knowledge of the learning algorithm; e.g., skilled attackers can manipulate data at test time to *evade* detection, or inject *poisoning* samples into the training data to mislead the learning algorithm and subsequently cause misclassification errors [7], [11], [25], [32], [34], [42], [43], [44], [45].

- *A. Demontis, M. Melis, B. Biggio, D. Maiorca, I. Corona, G. Giacinto, and F. Roli are with the Department of Electrical and Electronic Engineering, University of Cagliari, Piazza d'Armi, Cagliari 09123, Italy. E-mail: {ambra.demontis, marco.melis, battista.biggio, davide.maiorca, igino.corona, giacinto, roli}@diee.unica.it.*
- *D. Arp and K. Rieck are with the Institute of System Security, Technische Universität Braunschweig, Rebenring 56, Braunschweig 38106, Germany. E-mail: {arp, rieck}@sectubs.de.*

Manuscript received 9 Feb. 2016; revised 27 Mar. 2017; accepted 26 Apr. 2017. Date of publication 2 May 2017; date of current version 5 July 2019. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2017.2700270

In this paper, instead, we show that one can leverage machine learning to improve system security, by following an *adversary-aware* approach in which the machine-learning algorithm is designed from the *ground up* to be more resistant against evasion. We further show that designing adversary-aware learning algorithms according to this principle, as advocated in [9], [10], does not necessarily require one to trade classification accuracy in the absence of carefully-crafted attacks for improving security.

We consider Android malware detection as a case study for our approach. The relevance of this task is witnessed by the fact that Android has become the most popular mobile operating system, with more than a billion users around the world, while the number of malicious applications targeting them has also grown simultaneously: anti-virus vendors detect thousands of new malware samples daily, and there is still no end in sight [28], [50]. Here we focus our analysis on Drebin (Section 2), i.e., a machine-learning approach that relies on *static analysis* for an efficient detection of Android malware directly on the mobile device [3].

Notably, in this work we do not consider attacks that can completely defeat static analysis [31], like those based on packer-based encryption [47] and advanced code obfuscation [17], [23], [24], [35], [36]. The main reason is that such techniques may leave detectable traces, suggesting the use of a more appropriate system for classification; e.g., the presence of system routines that perform dynamic loading of libraries or classes, potentially hiding embedded malware, demands for the use of dynamic analysis for a more reliable classification. For this reason, in this paper we aim to improve the security of Drebin against *stealthier* attacks, i.e., carefully-crafted malware samples that evade detection without exhibiting significant evidence of manipulation.

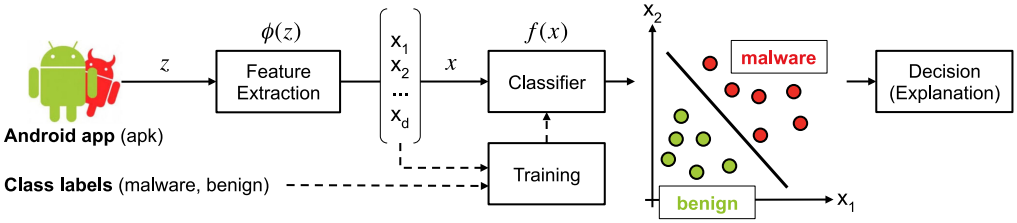

Fig. 1. A schematic representation of the architecture of Drebin. First, applications are represented as vector in a d-dimensional feature space. A linear classifier is then trained on an available set of labeled application, to discriminate between malware and benign applications. During classification, unseen applications are evaluated by the classifier. If its output $f(x) \geq 0$, they are classified as malware, and as benign otherwise. Drebin also provides an interpretation of its decision, by highlighting the most suspicious (or benign) features that contributed to the decision [3].

To perform a well-crafted security analysis of Drebin and, more generally, of Android malware detection tools against such attacks, we exploit an adversarial framework (Section 3) based on previous work on *adversarial machine learning* [4], [5], [9], [10], [25]. We focus on the definition of different classes of *evasion* attacks, corresponding to *attack scenarios* in which the attacker exhibits an increasing capability of manipulating the input data, and level of knowledge about the targeted system. To simulate evasion attacks in which the attacker does not exploit any knowledge of the targeted system, we consider some obfuscation techniques that are not specifically targeted against Drebin, by running an analysis similar to that reported in [30]. To this end, we make use of the commercial obfuscation tool `DexGuard`,[1] which has been originally designed to make reverse-engineering of benign applications more difficult. The obfuscation techniques exploited by this tool are discussed in detail in Section 4. Note that, even if considering obfuscation attacks is out of the scope of this work, `DexGuard` only partially obfuscates the content of Android applications. For this reason, the goal of this analysis is simply to empirically assess whether the static analysis performed by Drebin remains effective when Android applications are not thoroughly obfuscated, or when obfuscation is not targeted.

The main contribution of this work is the proposal of an *adversary-aware* machine-learning detector against *evasion attacks* (Section 5), inspired from the proactive design approach advocated in the area of adversarial machine learning [9], [10]. The secure machine-learning algorithm proposed in this paper is completely novel. With respect to previous techniques for *secure learning* [8], [15], [21], [26], it is able to retain computational efficiency and scalability on large datasets (as it exploits a linear classification function), while also being well-motivated from a more theoretical perspective. We empirically evaluate our method on real-world data (Section 6), including an adversarial security evaluation based on the simulation of the proposed evasion attacks. We show that our method outperforms state-of-the-art classification algorithms, including *secure* ones, without losing significant accuracy in the absence of well-crafted attacks, and can even guarantee some degree of robustness against `DexGuard`-based obfuscations. We finally discuss the main limitations of our work (Section 7), and future research challenges, including how to apply the proposed approach to other malware detection tasks (Section 8).

1. https://www.guardsquare.com/dexguard

2 Android Malware Detection

In this section, we give some background on Android applications. We then discuss Drebin and its main limitations.

2.1 Android Background

Android is the most used mobile operating system. Android applications are in the `apk` format, i.e., a zipped archive containing two files: the Android `manifest` and classes.dex. Additional `xml` and resource files are respectively used to define the application layout, and to provide additional functionality or multimedia content. As Drebin only analyzes the Android `manifest` and classes.dex files, below we provide a brief description of their characteristics.

Android Manifest. The `manifest` file holds information about the application structure. Such structure is organized in *application components*, i.e., parts of code that perform specific actions; e.g., one component might be associated to a screen visualized by the user (*activity*) or to the execution of audio in the background (*services*). The actions of each component are further specified through *filtered intents*; e.g., when a component sends data to other applications, or is invoked by a browser. Special types of components are *entry points*, i.e., activities, services and receivers that are loaded when requested by a specific filtered intent (e.g., an activity is loaded when an application is launched, and a service is activated when the device is turned on). The `manifest` also contains the list of *hardware components* and *permissions* requested by the application to work (e.g., Internet access).

Dalvik Bytecode **(`dexcode`).** The classes.dex file contains the compiled source code of an application. It contains all the user-implemented methods and classes. Classes.dex might contain specific API calls that can access sensitive resources such as personal contacts (*suspicious calls*). Moreover, it contains all system-related, *restricted API calls* whose functionality require *permissions* (e.g., using the Internet). Finally, this file can contain references to *network addresses* that might be contacted by the application.

2.2 Drebin

Drebin conducts multiple steps and can be executed directly on the mobile device, as it performs a lightweight static analysis of Android applications. The extracted features are used to embed applications into a high-dimensional vector space and train a classifier on a set of labeled data. An overview of the system architecture is given in Fig. 1. In the following, we describe the single steps in more detail.

TABLE 1
Overview of Feature Sets

Feature sets		
manifest	S_1	Hardware components
	S_2	Requested permissions
	S_3	Application components
	S_4	Filtered intents
dexcode	S_5	Restricted API calls
	S_6	Used permission
	S_7	Suspicious API calls
	S_8	Network addresses

2.2.1 Feature Extraction

Initially, Drebin performs a static analysis of a set of available Android applications,[2] to construct a suitable feature space. All features extracted by Drebin are presented as *strings* and organized in 8 different feature sets, as listed in Table 1. Android applications are then mapped onto the feature space as follows. Let us assume that an Android application (i.e., an `apk` file) is represented as an object $z \in \mathcal{Z}$, being $\mathcal{Z}$ the abstract space of all `apk` files. We then denote with $\Phi : \mathcal{Z} \mapsto \mathcal{X}$ a function that maps an `apk` file z to a d-dimensional feature vector $\boldsymbol{x} = (x^1, \ldots, x^{\mathsf{d}})^\top \in \mathcal{X} = \{0,1\}^{\mathsf{d}}$, where each feature is set to 1 (0) if the corresponding *string* is present (absent) in the `apk` file z. An application encoded in feature space may thus look like the following:

$$\boldsymbol{x} = \Phi(z) \mapsto \begin{pmatrix} \cdots \\ 0 \\ 1 \\ \cdots \\ 1 \\ 0 \\ \cdots \end{pmatrix} \begin{array}{l} \cdots \\ \texttt{permission :: SEND\_SMS} \\ \texttt{permission :: READ\_SMS} \\ \cdots \\ \texttt{api\_call :: getDeviceId} \\ \texttt{api\_call :: getSubscriberId} \\ \cdots \end{array} \begin{array}{l} \\ \Big\} S_2 \\ \\ \\ \Big\} S_5 \\ \\ \end{array} .$$

2.2.2 Learning and Classification

Once Android applications are represented as feature vectors, Drebin learns a *linear* Support Vector Machine (SVM) classifier [18], [41] to discriminate between the class of benign and malicious samples. Linear classifiers are generally expressed in terms of a linear function $f : \mathcal{X} \mapsto \mathbb{R}$, given as

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b\,, \tag{1}$$

where $\boldsymbol{w} \in \mathbb{R}^{\mathsf{d}}$ denotes the vector of *feature weights*, and $b \in \mathbb{R}$ is the so-called *bias*. These parameters, to be optimized during training, identify a hyperplane in feature space, which separates the two classes. During classification, unseen applications are then classified as malware if $f(\boldsymbol{x}) \geq 0$, and as benign otherwise.

During training, we are given a set of labeled samples $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{\mathsf{n}}$, where $\boldsymbol{x}_i$ denotes an application in feature space, and $y_i \in \{-1, +1\}$ its label, being -1 and $+1$ the benign and malware class, respectively. The SVM learning algorithm is then used to find the parameters $\boldsymbol{w}, b$ of Eq. (1), by solving the following optimization problem:

$$\min_{\boldsymbol{w},b} \mathcal{L}(\mathcal{D}, f) = \underbrace{\frac{1}{2}\boldsymbol{w}^\top \boldsymbol{w}}_{R(f)} + C \underbrace{\sum_{i=1}^{\mathsf{n}} \max(0, 1 - y_i f(\boldsymbol{x}_i))}_{L(f,\mathcal{D})}\,, \tag{2}$$

where $L(f, \mathcal{D})$ denotes a loss function computed on the training data (exhibiting higher values if samples in $\mathcal{D}$ are not correctly classified by f), $R(f)$ is a regularization term to avoid overfitting (i.e., to avoid that the classifier overspecializes its decisions on the training data, losing generalization capability on unseen data), and C is a trade-off parameter. As shown by the above problem, the SVM exploits an ℓ_2 regularizer on the feature weights and the so-called *hinge loss* as the loss function. This allows the SVM algorithm to learn a hyperplane that separates the two classes with the highest *margin* [18], [41]. Note that the above formulation is quite general, as it represents different learning algorithms, depending on the chosen regularizer and loss function [16].

2.3 Limitations and Open Issues

Although Drebin has shown to be capable of detecting malware with high accuracy, it exhibits intrinsic vulnerabilities that might be exploited by an attacker to evade detection. Since Drebin has been designed to run directly on the mobile device, its most obvious limitation is the lack of a dynamic analysis. Unfortunately, static analysis has clear limitations, as it is not possible to analyze malicious code that is downloaded or decrypted at runtime, or code that is thoroughly obfuscated [17], [23], [24], [31], [35], [36], [47]. For this reason, considering such attacks would be irrelevant for the scope of our work. Our focus is rather to understand and to improve the security properties of learning algorithms against specifically-targeted attacks, in which the amount of manipulations performed by the attacker is limited. The rationale is that the manipulated malware samples should not only evade detection, but it should also be difficult to detect traces of their adversarial manipulation. Although these limitations have been also discussed in [3], the effect of carefully-targeted attacks against Drebin has never been studied before. For this reason, in the following, we introduce an attack framework to provide a systematization of different, potential evasion attacks under limited adversarial manipulations. Then, we present a systematic evaluation of these attacks on Drebin, and a novel learning algorithm to alleviate their effects in practice.

3 Attack Model and Scenarios

To perform a thorough security assessment of learning-based malware detection systems, we rely upon an attack model originally defined in [9], [10]. It is grounded on the popular taxonomy of Barreno et al. [4], [5], [25], which categorizes potential attacks against machine-learning algorithms along three axes: *security violation*, *attack specificity* and *attack influence*. The attack model exploits this taxonomy to define a number of potential attack scenarios that may be incurred by the system during operation, in terms of explicit assumptions on the attacker's goal, knowledge of the system, and capability of manipulating the input data.

3.1 Attacker's Goal

It is defined in terms of the desired security violation and the so-called attack specificity.

2. We use here a modified version of Drebin that performs a static analysis based on the `Androguard` tool, available at: https://github.com/androguard/androguard.

Security Violation. Security can be compromised by violating system *integrity*, if malware samples are undetected; system *availability*, if benign samples are misclassified as malware; or *privacy*, if the system leaks confidential information about its users.

Attack Specificity. It can be *targeted* or *indiscriminate*, depending on whether the attacker is interested in having some specific samples misclassified (e.g., a specific malware sample to infect a particular device), or if any misclassified sample meets her goal (e.g., if the goal is to launch an indiscriminate attack campaign).

3.2 Attacker's Knowledge

The attacker may have different levels of knowledge of the targeted system [4], [5], [9], [10], [25], [43]. In particular, she may know completely, partially, or do not have any information at all about: (i) the training data $\mathcal{D}$; (ii) the feature extraction/selection algorithm Φ, and the corresponding feature set $\mathcal{X}$, i.e., how features are computed from data, and selected; (iii) the learning algorithm $\mathcal{L}(\mathcal{D}, f)$, along with the decision function $f(\boldsymbol{x})$ (Eq. (1)) and, potentially, even its (trained) parameters $\boldsymbol{w}$ and b. In some applications, the attacker may also exploit feedback on the classifier's decisions to improve her knowledge of the system, and, more generally, her attack strategy [5], [9], [10], [25].

3.3 Attacker's Capability

It consists of defining the *attack influence* and how the attacker can manipulate data.

Attack Influence. It can be *exploratory*, if the attacker only manipulates data at test time, or *causative*, if she can also contaminate the training data (e.g., this may happen if a system is periodically retrained on data collected during operation that can be modified by an attacker) [5], [10], [25].

Data Manipulation. It defines how samples (and features) can be modified, according to application-specific constraints; e.g., which feature values can be incremented or decremented without compromising the exploitation code embedded in the *apk* file. In many cases, these constraints can be encoded in terms of distances in feature space, computed between the source malware data and its manipulated versions [7], [15], [19], [21], [29], [40]. We refer the reader to Section 3.6 for a discussion on how Drebin features can be modified.

3.4 Attack Strategy

The attack strategy defines how the attacker implements her activities, based on the hypothesized goal, knowledge, and capabilities. To this end, we characterize the attacker's knowledge in terms of a space Θ that encodes knowledge of the data $\mathcal{D}$, the feature space $\mathcal{X}$, and the classification function f. Accordingly, we can represent the scenario in which the attacker has perfect knowledge of the attacked system as a vector $\boldsymbol{\theta} = (\mathcal{D}, \mathcal{X}, f) \in \Theta$. We characterize the attacker's capability by assuming that an initial set of samples $\mathcal{A}$ is given, and that it is modified according to a space of possible modifications $\Omega(\mathcal{A})$. Given the attacker's knowledge $\boldsymbol{\theta} \in \Theta$ and a set of manipulated attacks $\mathcal{A}' \in \Omega(\mathcal{A}) \subseteq \mathcal{Z}$, the attacker's goal can be characterized in terms of an objective function $\mathcal{W}(\mathcal{A}', \boldsymbol{\theta}) \in \mathbb{R}$ which evaluates the extent to which the manipulated attacks $\mathcal{A}'$ meet the attacker's goal. The optimal attack strategy can be thus given as

$$\mathcal{A}^* = \arg\max_{\mathcal{A}' \in \Omega(\mathcal{A})} \mathcal{W}(\mathcal{A}'; \boldsymbol{\theta}) \,. \tag{3}$$

Under this formulation, one can characterize different attack scenarios. The two main ones often considered in adversarial machine learning are referred to as classifier *evasion* and *poisoning* [4], [5], [7], [9], [10], [11], [25], [45]. In the remainder of this work we focus on *classifier evasion*, while we refer the reader to [10], [45] for further details on *classifier poisoning*.

3.5 Evasion Attacks

In an evasion attack, the attacker manipulates malicious samples at test time to have them misclassified as benign by a trained classifier, without having *influence* over the training data. The attacker's goal thus amounts to violating system *integrity*, either with a *targeted* or with an *indiscriminate* attack, depending on whether the attacker is targeting a specific machine or running an indiscriminate attack campaign. More formally, evasion attacks can be written as

$$\boldsymbol{z}^* = \arg\min_{\boldsymbol{z}' \in \Omega(\boldsymbol{z})} \hat{f}(\Phi(\boldsymbol{z}')) = \arg\min_{\boldsymbol{z}' \in \Omega(\boldsymbol{z})} \hat{\boldsymbol{w}}^\top \boldsymbol{x}' \,, \tag{4}$$

where $\boldsymbol{x}' = \Phi(\boldsymbol{z}')$ is the feature vector associated to the modified attack sample $\boldsymbol{z}'$, and $\hat{\boldsymbol{w}}$ is the weight vector estimated by the attacker (e.g., from the surrogate classifier $\hat{f}$). With respect to Eq. (3), one can consider here one sample at a time, as they can be independently modified.

The above equation essentially tells the attacker which features should be modified to maximally decrease the value of the classification function, i.e., to maximize the probability of evading detection [7], [10]. Note that, depending on the manipulation constraints $\Omega(\boldsymbol{z})$ (e.g., if the feature values are bounded), the set of features to be manipulated is generally different for each malicious sample.

In the following, we consider different evasion scenarios, according to the framework discussed in the previous sections. In particular, we discuss five distinct attack scenarios, sorted for increasing level of attacker's knowledge. Note that, when the attacker knows more details of the targeted system, her estimate of the classification function becomes more reliable, thus facilitating the evasion task (in the sense of requiring less manipulations to the malware samples).

3.5.1 Zero-Effort Attacks

This is the standard scenario in which malware data is neither obfuscated nor modified at all. From the viewpoint of the attacker's knowledge, this scenario is characterized by an empty knowledge-parameter vector $\boldsymbol{\theta} = ()$.

3.5.2 DexGuard-Based Obfuscation Attacks

As another attack scenario in which the attacker does not exploit any knowledge of the attacked system, for which $\boldsymbol{\theta} = ()$, we consider a setting similar to that reported in [30]. In particular, we assume that the attacker attempts to evade detection by performing invasive code transformations on the `classes.dex` file, using the commercial Android obfuscation tool `DexGuard`. Note that this tool is designed to ensure protection against disassembling/decompiling attempts in benign applications, and not to obfuscate the presence of

malicious code; thus, despite the introduction of many changes in the executable code, it is not clear whether and to what extent the obfuscations implemented by this tool may be effective against a learning-based malware detector like Drebin, i.e., how they will affect the corresponding feature values and classification output. The obfuscations implemented by `DexGuard` are described more in detail in Section 4.

3.5.3 Mimicry Attacks

Under this scenario, the attacker is assumed to be able to collect a surrogate dataset including malware and benign samples, and to know the feature space. Accordingly, $\boldsymbol{\theta} = (\hat{\mathcal{D}}, \mathcal{X})$. In this case, the attack strategy amounts to manipulating malware samples to make them as close as possible to the benign data (in terms of conditional probability distributions or, alternatively, distance in feature space). To this end, in the case of Drebin (which uses binary feature values), we can assume that the attacker still aims to minimize Eq. (4), but estimates each component of $\hat{w}$ independently for each feature as $\hat{w}_k = p(\hat{x}_k = 1|y = +1) - p(\hat{x}_k = 1|y = -1)$, $k = 1, \ldots, \mathsf{d}$. This will indeed induce the attacker to add (remove) first features which are more frequently present (absent) in benign files, making the probability distribution of malware samples closer to that of the benign data. It is worth finally remarking that this is a more sophisticated mimicry attack than those commonly used in practice, in which an attacker is usually assumed to merge a malware application with a benign one [43], [50].

3.5.4 Limited-Knowledge (LK) Attacks

In addition to the previous case, here the attacker knows the learning algorithm $\mathcal{L}$ used by the targeted system, and can learn a surrogate classifier on the available data. The knowledge-parameter vector can be thus encoded as $\boldsymbol{\theta} = (\hat{\mathcal{D}}, \mathcal{X}, \hat{f})$, being $\hat{f}$ the surrogate classifier used to approximate the true f. In this case, the attacker exploits the estimate of $\hat{w}$ obtained from the surrogate classifier $\hat{f}$ to construct the evasion samples, according to Eq. (4).

3.5.5 Perfect-Knowledge (PK) Attacks

This is the worst-case setting in which also the targeted classifier is known to the attacker, i.e., $\boldsymbol{\theta} = (\mathcal{D}, \mathcal{X}, f)$. Although it is not very likely to happen in practice that the attacker gets to know even the trained classifier's parameters (i.e., $\boldsymbol{w}$ and b in Eq. (1)), this setting is particularly interesting as it provides an upper bound on the performance degradation incurred by the system under attack, and can be used as reference to evaluate the effectiveness of the system under the other simulated attack scenarios.

3.6 Malware Data Manipulation

As stated in Section 3.3, one has to discuss how the attacker can manipulate malware applications to create the corresponding evasion attack samples. To this end, we consider two main settings in our evaluation, detailed below.

Feature Addition. Within this setting, the attacker can independently inject (i.e., set to 1) every feature.

Feature Addition and Removal. This scenario simulates a more powerful attacker that can inject every feature, and also remove (i.e., set to 0) features from the `dexcode`.

These settings are motivated by the fact that malware has to be manipulated to evade detection, but its semantics and intrusive functionality must be preserved. In this respect, *feature addition* is generally a safe operation, in particular, when injecting `manifest` features (e.g., adding permissions does not influence any existing application functionality). With respect to the `dexcode`, one may also safely introduce information that is not actively executed, by adding code after `return` instructions (*dead code*) or with methods that are never called by any `invoke` type instructions. Listing 1 shows an example where a URL feature is introduced by adding a method that is never invoked in the code.

Listing 1. Smali code to add a URL feature

```
.method public addUrlFeature()V
  .locals 2
  ;const-string v1, "http://www.example.com"
  invoke-direct {v0, v1},
  Ljava/net/URL;-><init>(Ljava/lang/String;)V
  return-void
.end method
```

However, this only applies when such information is not directly executed by the application, and could be stopped at the parsing level by analyzing only the methods belonging to the application *call graph*. In this case, the attacker would be enforced to change the executed code, and this requires considering additional and stricter constraints. For example, if she wants to add a suspicious API call to a `dexcode` method that is executed by the application, she should adopt virtual machine registers that have not been used before by the application. Moreover, the attacker should pay attention to possible artifacts or undesired functionalities that are brought by the injected calls, which may influence the semantics of the original program. Accordingly, injecting a large number of features may not always be feasible.

Feature removal is even a more complicated operation. Removing permissions from the `manifest` is not possible, as this would limit the application functionality. The same holds for intent filters. Some application component names can be changed but, as stated in Section 4, this operation is not easy to be automatically performed: the attacker must ensure that the application component names in the `dexcode` are changed accordingly, and must not modify any of the entry points. Furthermore, the feasible changes may only slightly affect the whole `manifest` structure (as shown in our experiments with automated obfuscation tools). With respect to the `dexcode`, multiple ways can be exploited to remove its features; e.g., it is possible to hide IP addresses (if they are stored as strings) by encrypting them with the introduction of additional functions, and decrypting them at runtime. Of course, this should be done by avoiding the addition of features that are already used by the system (e.g., function calls that are present in the training data).

With respect to suspicious and restricted API calls, the attacker should encrypt the method or the class invoking them. However, this could introduce other calls that might increase the suspiciousness of the application. Moreover, one mistake at removing such API references might completely destroy the application functionality. The reason is that Android uses a *verification* system to check the

integrity of an application during execution (e.g., it will close the application, if a register passed as a parameter to an API call contains a wrong type), and chances of compromising this behavior increase if features are deleted carelessly.

For the aforementioned reasons, performing a fine-grained evasion attack that changes a lot of features may be very difficult in practice, without compromising the malicious application functionality. In addition, another problem for the attacker is getting to know precisely which features should be added or removed, which makes the construction of evasion attack samples even more complicated.

4 DexGuard-Based Obfuscation Attacks

Although commercial obfuscators are designed to protect benign applications against reverse-engineering attempts, it has been recently shown that they can also be used to evade anti-malware detection systems [30]. We thus use `DexGuard`, a popular obfuscator for Android, to simulate attacks in which no specific knowledge of the targeted system is exploited, as discussed in Section 3.5.2. Recall that, although considering obfuscation attacks is out of the scope of this work, the obfuscation techniques implemented by `DexGuard` do not completely obfuscate the code. For this reason, we aim to understand whether this may make static analysis totally ineffective, and how it affects our strategy to improve classifier security. A brief description of the `DexGuard`-based obfuscation attacks is given below.

Trivial Obfuscation. This strategy changes the names of *implemented* application packages, classes, methods and fields, by replacing them with random characters. Trivial obfuscation also performs negligible modifications to some `manifest` features by renaming some application components that are *not* entry points (see Section 2.1). As the application functionality must be preserved, Trivial obfuscation does not rename any system API or method imported from native libraries. Given that Drebin mainly extracts information from system APIs, we expect that its detection capability will be only barely affected by this obfuscation.

String Encryption. This strategy encrypts strings defined in the `dexcode` with the instruction `const-string`. Such strings can be visualized during the application execution, or may be used as variables. Thus, even if they are retrieved through an identifier, their value must be preserved during the program execution. For this reason, an additional method is added to decrypt them at runtime, when required. This obfuscation tends to remove URL features (S8) that are stored as strings in the `dexcode`. Features corresponding to the decryption routines extracted by Drebin (S7) are instead not affected, as the decryption routines added by `DexGuard` do not belong to the system APIs.

Reflection. This obfuscation technique uses the Java Reflection API to replace `invoke-type` instructions with calls that belong to the `Java.lang.Reflect` class. The main effect of this action is destroying the application call graph. However, this technique does not affect the system API names, as they do not get encrypted during the process. It is thus reasonable to expect that most of the features extracted by Drebin will remain unaffected.

Class Encryption. This is the most invasive obfuscation strategy, as it encrypts all the application classes, except entry-point ones (as they are required to load the application externally). The encrypted classes are decrypted at runtime by routines that are added during the obfuscation phase. Worth noting, the class encryption performed by `DexGuard` does not completely encrypt the application. For example, classes belonging to the API components contained in the `manifest` are not encrypted, as this would most likely compromise the application functionality. For the same reason, the `manifest` itself is preserved. Accordingly, it still possible to extract static features using Drebin, and analyze the application. Although out of the scope of our work, it is still worth remarking here that using packers (e.g., [47]) to perform full dynamic loading of the application classes might completely evade static analysis.

Combined Obfuscations. The aforementioned strategies can also be combined to produce additional obfuscation techniques. As in [30], we will consider three additional techniques in our experiments, by respectively combining (i) trivial and string encryption, (ii) adding reflection to them, and (iii) adding class encryption to the former three.

5 Adversarial Detection

In this section, we introduce an adversary-aware approach to improve the robustness of Drebin against carefully-crafted data manipulation attacks. As for Drebin, we aim to develop a simple, lightweight and scalable approach. For this reason, the use of non-linear classification functions with computationally-demanding learning procedures is not suitable for our application setting. We have thus decided to design a linear classification algorithm with improved security properties, as detailed in the following.

5.1 Securing Linear Classification

As in previous work [8], [26], we aim to improve the security of our linear classification system by enforcing learning of more *evenly-distributed* feature weights, as this would intuitively require the attacker to manipulate more features to evade detection. Recall that, as discussed in Section 3.6, if a large number of features has to be manipulated to evade detection, it may not even be possible to construct the corresponding malware sample without compromising its malicious functionality. With respect to the work in [8], [26], where different heuristic implementations were proposed to improve the so-called *evenness* of feature weights (see Section 6), we propose here a more principled approach, derived from the idea of *bounding* classifier sensitivity to feature changes.

We start by defining a measure of *classifier sensitivity* as

$$\Delta f(\boldsymbol{x}, \boldsymbol{x}') = \frac{f(\boldsymbol{x}) - f(\boldsymbol{x}')}{\|\boldsymbol{x} - \boldsymbol{x}'\|} = \frac{\boldsymbol{w}^\top (\boldsymbol{x} - \boldsymbol{x}')}{\|\boldsymbol{x} - \boldsymbol{x}'\|}\,, \tag{5}$$

which evaluates the decrease of f when a malicious sample $\boldsymbol{x}$ is manipulated as $\boldsymbol{x}'$, with respect to the required amount of modifications, given by $\|\boldsymbol{x} - \boldsymbol{x}'\|$.

Let us assume now, without loss of generality, that $\boldsymbol{w}$ has unary ℓ_1-norm and that features are normalized in $[0, 1]$.[3]

3. Note that this is always possible without affecting system performance, by dividing f by $\|\boldsymbol{w}\|_1$, and normalizing feature values on a compact domain before classifier training.

We also assume that, for simplicity, the ℓ_1-norm is used to evaluate $\|\boldsymbol{x} - \boldsymbol{x}'\|$. Under these assumptions, it is not difficult to see that $\Delta f \in \left[\frac{1}{\mathsf{d}}, 1\right]$, where the minimum is attained for equal absolute weight values (regardless of the amount of modifications made to $\boldsymbol{x}$), and the maximum is attained when only one weight is not null, confirming the intuition that more *evenly-distributed* feature weights should improve classifier security under attack. This can also be shown by selecting $\boldsymbol{x}, \boldsymbol{x}'$ to maximize $\Delta f(\boldsymbol{x}, \boldsymbol{x}')$

$$\Delta f(\boldsymbol{x}, \boldsymbol{x}') \leq \frac{1}{\mathsf{K}} \sum_{k=1}^{\mathsf{K}} |w_{(k)}| \leq \max_{j=1,\ldots,\mathsf{d}} |w_j| = \|\boldsymbol{w}\|_\infty . \quad (6)$$

Here, $\mathsf{K} = \|\boldsymbol{x} - \boldsymbol{x}'\|$ corresponds to the number of modified features and $|w_{(1)}|, \ldots, |w_{(\mathsf{d})}|$ denote the weights sorted in descending order of their absolute values, such that we have $|w_{(1)}| \geq \cdots \geq |w_{(\mathsf{d})}|$. The last inequality shows that, to minimize classifier sensitivity to feature changes, one can minimize the ℓ_∞-norm of $\boldsymbol{w}$. This in turn tends to promote solutions which exhibit the same absolute weight values (a well-known effect of ℓ_∞ regularization [13]).

This is a very interesting result which has never been pointed out in the field of adversarial machine learning. We have shown that regularizing our learning algorithm by penalizing the ℓ_∞ norm of the feature weights $\boldsymbol{w}$ can improve the *security* of linear classifiers, yielding classifiers with more *evenly-distributed* feature weights. This has only been intuitively motivated in previous work, and implemented with heuristic approaches [8], [26]. As we will show in Section 6, being derived from a more principled approach, our method is not only capable of finding more *evenly-distributed* feature weights with respect to the heuristic approaches in [8], [26], but it is also able to outperform them in terms of security.

It is also worth noting that our approach preserves *convexity* of the objective function minimized by the learning algorithm. This gives us the possibility of deriving computationally-efficient training algorithms with (potentially strong) convergence guarantees. As an alternative to considering an additional term to the learner's objective function $\mathcal{L}$, one can still control the ℓ_∞-norm of $\boldsymbol{w}$ by adding a box constraint on it. This is a well-known property of convex optimization [13]. As we may need to apply different upper and lower bounds to different feature sets, depending on how their values can be manipulated, we prefer to follow the latter approach.

5.2 Secure SVM Learning Algorithm

According to the previous discussion, we define our Secure SVM learning algorithm (Sec-SVM) as

$$\min_{\boldsymbol{w},b} \ \frac{1}{2}\boldsymbol{w}^\top \boldsymbol{w} + C \sum_{i=1}^{\mathsf{n}} \max(0, 1 - y_i f(\boldsymbol{x}_i)) , \quad (7)$$

$$\text{s.t.} \ \ w_k^{\text{lb}} \leq w_k \leq w_k^{\text{ub}} , \ k = 1, \ldots, \mathsf{d} . \quad (8)$$

Note that this optimization problem is identical to Problem (2), except for the presence of a box constraint on $\boldsymbol{w}$. The lower and upper bounds on $\boldsymbol{w}$ are defined by the vectors $\boldsymbol{w}^{\text{lb}} = (w_1^{\text{lb}}, \ldots, w_{\mathsf{d}}^{\text{lb}})$ and $\boldsymbol{w}^{\text{ub}} = (w_1^{\text{ub}}, \ldots, w_{\mathsf{d}}^{\text{ub}})$, which should be selected with a suitable procedure (see Section 5.3). For notational convenience, in the sequel we will also denote the constraint given by Eq. (8) compactly as $\boldsymbol{w} \in \mathcal{W} \subseteq \mathbb{R}^{\mathsf{d}}$.

Algorithm 1. Sec-SVM Learning Algorithm

Input: $\mathcal{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^{\mathsf{n}}$, the training data; C, the regularization parameter; $\boldsymbol{w}^{\text{lb}}, \boldsymbol{w}^{\text{ub}}$, the lower and upper bounds on $\boldsymbol{w}$; $|\mathcal{S}|$, the size of the sample subset used to approximate the subgradients; $\eta^{(0)}$, the initial gradient step size; $s(t)$, a decaying function of t; and $\varepsilon > 0$, a small constant.

Output: $\boldsymbol{w}, b$, the trained classifier's parameters.

Set iteration count $t \leftarrow 0$.
Randomly initialize $\boldsymbol{v}^{(t)} = (\boldsymbol{w}^{(t)}, b^{(t)}) \in \mathcal{W} \times \mathbb{R}$.
Compute the objective function $\mathcal{L}(\boldsymbol{v}^{(t)})$ using Eq. (7).
repeat
  Compute $(\nabla_{\boldsymbol{w}}\mathcal{L}, \nabla_b\mathcal{L})$ using Eqs. (9) and (10).
  Increase the iteration count $t \leftarrow t + 1$.
  Set $\eta^{(t)} \leftarrow \gamma\, \eta^{(0)} s(t)$, performing a line search on γ.
  Set $\boldsymbol{w}^{(t)} \leftarrow \boldsymbol{w}^{(t-1)} - \eta^{(t)} \nabla_w \mathcal{L}$.
  Project $\boldsymbol{w}^{(t)}$ onto the feasible (box) domain $\mathcal{W}$.
  Set $b^{(t)} \leftarrow b^{(t-1)} - \eta^{(t)} \nabla_b \mathcal{L}$.
  Set $\boldsymbol{v}^{(t)} = (\boldsymbol{w}^{(t)}, b^{(t)})$.
  Compute the objective function $\mathcal{L}(\boldsymbol{v}^{(t)})$ using Eq. (7).
until $|\mathcal{L}(\boldsymbol{v}^{(t)}) - \mathcal{L}(\boldsymbol{v}^{(t-1)})| < \varepsilon$
return: $\boldsymbol{w} = \boldsymbol{w}^{(t)}$, and $b = b^{(t)}$.

The corresponding learning algorithm is given as Algorithm 1. It is a constrained variant of Stochastic Gradient Descent (SGD) that also considers a simple line-search procedure to tune the gradient step size during the optimization. SGD is a lightweight gradient-based algorithm for efficient learning on very large-scale datasets, based on approximating the subgradients of the objective function using a single sample or a small subset of the training data, randomly chosen at each iteration [12], [49]. In our case, the subgradients of the objective function (Eq. (7)) are given as

$$\nabla_{\boldsymbol{w}}\mathcal{L} \ \approx \ \boldsymbol{w} + C \sum_{i \in \mathcal{S}} \nabla_\ell^i \boldsymbol{x}_i , \quad (9)$$

$$\nabla_b\mathcal{L} \ \approx \ C \sum_{i \in \mathcal{S}} \nabla_\ell^i , \quad (10)$$

where $\mathcal{S}$ denotes the subset of the training samples used to compute the approximation, and ∇_ℓ^i is the gradient of the hinge loss with respect to $f(\boldsymbol{x}_i)$, which equals $-y_i$, if $y_i f(\boldsymbol{x}_i) < 1$, and 0 otherwise.

One crucial issue to ensure quick convergence of SGD is the choice of the initial gradient step size $\eta^{(0)}$, and of a proper *decaying function* $s(t)$, i.e., a function used to gradually reduce the gradient step size during the optimization process. As suggested in [12], [49], these parameters should be chosen based on preliminary experiments on a subset of the training data. Common choices for the function $s(t)$ include linear and exponential decaying functions.

We conclude this section by pointing out that our formulation is quite general; one may indeed select different combinations of loss and regularization functions to train different, secure variants of other linear classification algorithm. Our Sec-SVM learning algorithm is only an instance that considers the hinge loss and ℓ_2 regularization, as the standard SVM [18], [41]. It is also worth remarking that, as the lower and upper bounds become smaller in absolute value, our method tends to yield (*dense*) solutions with

weights equal to the upper or to the lower bound. A similar effect is obtained when minimizing the ℓ_∞ norm directly [13].

We conclude from this analysis that there is an implicit trade-off between *security* and *sparsity*: while a sparse learning model ensures an efficient description of the learned decision function, it may be easily circumvented by just manipulating a few features. By contrast, a secure learning model relies on the presence of many, possibly redundant, features that make it harder to evade the decision function, yet at the price of a dense representation.

5.3 Parameter Selection

To tune the parameters of our classifiers, as suggested in [10], [48], one should not only optimize accuracy on a set of collected data, using traditional performance evaluation techniques like cross validation or bootstrapping. More properly, one should optimize a trade-off between accuracy and *security*, by accounting for the presence of potential, unseen attacks during the validation procedure. Here we optimize this trade-off, denoted with $r(f_{\boldsymbol{\mu}}, \mathcal{D})$, as

$$\boldsymbol{\mu}^\star = \arg\max_{\boldsymbol{\mu}} r(f_{\boldsymbol{\mu}}, \mathcal{D}) = A(f_{\boldsymbol{\mu}}, \mathcal{D}) + \lambda S(f_{\boldsymbol{\mu}}, \mathcal{D}), \quad (11)$$

where we denote with $f_{\boldsymbol{\mu}}$ the classifier learned with parameters $\boldsymbol{\mu}$ (e.g., for our Sec-SVM, $\boldsymbol{\mu} = \{C, \boldsymbol{w}^{\text{lb}}, \boldsymbol{w}^{\text{ub}}\}$), with A a measure of classification accuracy in the absence of attack (estimated on $\mathcal{D}$), with S an estimate of the classifier security under attack (estimated by simulating attacks on $\mathcal{D}$), and with λ a given trade-off parameter.

Classifier security can be evaluated by considering distinct attack settings, or a different amount of modifications to the attack samples. In our experiments, we will optimize security in a worst-case scenario, i.e., by simulating a PK evasion attack with both feature addition and removal. We will then average the performance under attack over an increasing number of modified features $m \in [1, \mathsf{M}]$. More specifically, we will measure security as

$$S = \frac{1}{\mathsf{M}} \sum_{m=1}^{\mathsf{M}} A(f_{\boldsymbol{\mu}}, \mathcal{D}'_k), \quad (12)$$

where $\mathcal{D}'_k$ is obtained by modifying a maximum of m features in each malicious sample in the *validation set*,[4] as suggested by the PK evasion attack strategy.

6 Experimental Analysis

In this section, we report an experimental evaluation of our proposed secure learning algorithm (Sec-SVM) by testing it under different evasion scenarios (see Section 3.5).

Classifiers. We compare our Sec-SVM approach with the standard Drebin implementation (denoted with SVM), and with a previously-proposed technique that improves security of linear classifiers by using a Multiple Classifier System (MCS) architecture to obtain a linear classifier with more evenly-distributed feature weights [8], [26]. To this end, multiple linear classifiers are learned by sampling uniformly from the training set (a technique known as *bagging* [14]) and by randomly subsampling the feature set, as suggested by the *random subspace method* [22]. The classifiers are then combined by averaging their outputs, which is equivalent to using a linear classifier whose weights and bias are the average of the weights and biases of the base classifiers, respectively. With this simple trick, the computational complexity at test time remains thus equal to that of a single linear classifier [8]. As we use linear SVMs as the base classifiers, we denote this approach with MCS-SVM. We finally consider a version of our Sec-SVM trained using only `manifest` features that we call Sec-SVM (M). The reason is to verify whether considering only features, which can not removed, limits closely mimicking benign data and thereby yields a more secure system.

Datasets. In our experiments, we use two distinct datasets. The first (referred to as *Drebin*) includes the data used in [3], and consists of 121,329 benign applications and 5,615 malicious samples, labeled using the VirusTotal service. A sample is labeled as malicious if it is detected by at least five anti-virus scanners, whereas it is labeled as benign if no scanner flagged it as malware. The second (referred to as *Contagio*) includes the data used in [30], and consists of about 1,500 malware samples, obtained from the *MalGenome*[5] and the *Contagio Mobile Minidump*[6] datasets. Such samples have been obfuscated with the seven obfuscation techniques described in Section 4, yielding a total of about 10,500 samples.

Training-Test Splits. We average our results on 10 independent runs. In each repetition, we randomly select 60,000 applications from the *Drebin* dataset, and split them into two equal sets of 30,000 samples each, respectively used as the training set and the surrogate set (as required by the LK and mimicry attacks discussed in Section 3.5). As for the test set, we use all the remaining samples from *Drebin*. In some attack settings (detailed below), we replace the malware data from *Drebin* in each test set with the malware samples from *Contagio*. This enables us to evaluate the extent to which a classifier (trained on some data) preserves its performance in detecting malware from *different* sources.[7]

Feature Selection. When running Drebin on the given datasets, more than one million of features are found. For computational efficiency, we retain the most discriminant d' features, for which $|p(x_k = 1|y = +1) - p(x_k = 1|y = -1)|$, $k = 1, \ldots, \mathsf{d}$, exhibits the highest values (estimated on training data). In our case, using only $\mathsf{d}' = 10,000$ features does not significantly affect the accuracy of Drebin. This is consistent with the recent findings in [37], as it is shown that only a very small fraction of features is significantly discriminant, and usually assigned a non-zero weight by Drebin (i.e., by the SVM learning algorithm). For the same reason, the sets of selected features turned out to be the same in each run. Their sizes are reported in Table 2.

Parameter Setting. We run some preliminary experiments on a subset of the training set and noted that changing C did not have a significant impact on classification accuracy for all

4. Note that, as in standard performance evaluation techniques, data is split into distinct training-validation pairs, and then performance is averaged on the distinct validation sets. As we are considering *evasion attacks*, training data is not affected during the attack simulation, and only malicious samples in the validation set are thus modified.

5. http://www.malgenomeproject.org/

6. http://contagiominidump.blogspot.com/

7. Note however that a number of malware samples in *Contagio* are also included in the *Drebin* dataset.

TABLE 2
Number of Features in Each Set for SVM, Sec-SVM, and MCS-SVM

Feature set sizes					
manifest	S_1	13 (21)	dexcode	S_5	147 (0)
	S_2	152 (243)		S_6	37 (0)
	S_3	2,542 (8,904)		S_7	3,029 (0)
	S_4	303 (832)		S_8	3,777 (0)

Feature set sizes for the Sec-SVM (M) using only `manifest` *features are reported in brackets. For all classifiers, the total number of selected features is* $\mathsf{d}' = 10,000$.

the SVM-based classifiers (except for higher values, which cause overfitting). Thus, also for the sake of a fair comparison among different SVM-based learners, we set $C = 1$ for all classifiers and repetitions. For the MCS-SVM classifier, we train 50 base linear SVMs on random subsets of 80 percent of the training samples and 50 percent of the features, as this ensures a sufficient diversification of the base classifiers, providing more evenly-distributed feature weights. The bounds of the Sec-SVM are selected through a five-fold cross-validation, following the procedure explained in Section 5.3. In particular, we set each element of $\boldsymbol{w}^{\text{ub}}$ ($\boldsymbol{w}^{\text{lb}}$) as w^{ub} (w^{lb}), and optimize the two scalar values $(w^{\text{ub}}, w^{\text{lb}}) \in \{0.1, 0.5, 1\} \times \{-1, -0.5, -0.1\}$. As for the performance measure $A(f_{\boldsymbol{\mu}}, \mathcal{D})$ (Eq. (11)), we consider the Detection Rate (DR) at 1 percent False Positive Rate (FPR), while the security measure $S(f_{\boldsymbol{\mu}}, \mathcal{D})$ is simply given by Eq. (12). We set $\lambda = 10^{-2}$ in Eq. (11) to avoid worsening the detection of both benign and malware samples in the absence of attack to an unnecessary extent. Finally, as explained in Section 5.2, the parameters of Algorithm 1 are set by running it on a subset of the training data, to ensure quick convergence, as $\eta^{(0)} = 0.5, \gamma \in \{10, 20, \ldots, 70\}$ and $s(t) = 2^{-0.01t}/\sqrt{\mathsf{n}}$.

Evasion Attack Algorithm. We discuss here the algorithm used to implement our advanced evasion attacks. For linear classifiers with binary features, the solution to Problem (4) can be found as follows. First, the estimated weights $\hat{\boldsymbol{w}}$ have to be sorted in descending order of their absolute values, along with the feature values $\boldsymbol{x}$ of the initial malicious sample. This means that, if the sorted weights and features are denoted respectively with $\hat{w}_{(1)}, \ldots, \hat{w}_{(\mathsf{d})}$ and $x_{(1)}, \ldots, x_{(\mathsf{d})}$, then $|\hat{w}_{(1)}| \geq \ldots \geq |\hat{w}_{(\mathsf{d})}|$. Then, for $k = 1, \ldots, \mathsf{d}$: if $x_{(k)} = 1$ and $\hat{w}_{(k)} > 0$ (and the feature is not in the `manifest` sets S1-S4), then $x_{(k)}$ is set to zero; if $x_{(k)} = 0$ and $\hat{w}_{(k)} < 0$, then $x_{(k)}$ is set to one; else $x_{(k)}$ is left unmodified.

If the maximum number of modified features has been reached, the for loop is clearly stopped in advance.

6.1 Experimental Results

We present our results by reporting the performance of the given classifiers against (i) zero-effort attacks, (ii) obfuscation attacks, and (iii) advanced evasion attacks, including PK, LK and mimicry attacks, with both feature addition, and feature addition and removal (see Sections 3.5 and 3.6).

Zero-Effort Attacks. Results for the given classifiers in the absence of attack are reported in the ROC curves of Fig. 2. They report the Detection Rate (DR, i.e., the fraction of correctly-classified malware samples) as a function of the False Positive Rate (FPR, i.e., the fraction of misclassified benign

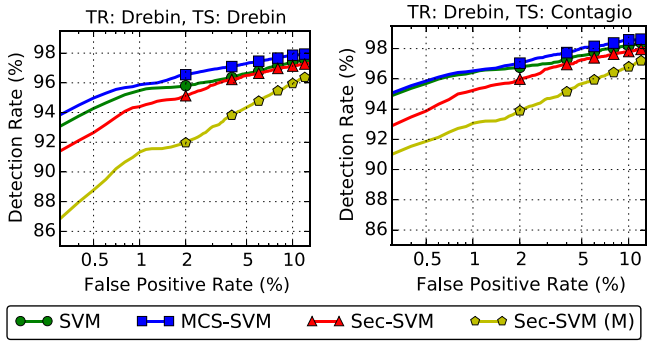

Fig. 2. Mean ROC curves on *Drebin* (*left*) and *Contagio* (*right*) data, for classifiers trained on *Drebin* data.

samples) for each classifier. We consider two different cases: (i) using both training and test samples from *Drebin* (*left* plot); and (ii) training on *Drebin* and testing on *Contagio* (*right* plot), as previously discussed. Notably, MCS-SVM achieves the highest DR (higher than 96 at 1 percent FPR) in both settings, followed by SVM and Sec-SVM, which only slightly worsen the DR. Sec-SVM (M) performs instead significantly worse. In Fig. 3, we also report the absolute weight values (sorted in descending order) of each classifier, to show that Sec-SVM classifiers yield more evenly-distributed weights, also with respect to MCS-SVM.

DexGuard-Based Obfuscation Attacks. The ROC curves reported in Fig. 4 show the performance of the given classifiers, trained on *Drebin*, against the `DexGuard`-based obfuscation attacks (see Sections 3.5.2 and 4) on the *Contagio* malware. Here, Sec-SVM performs similarly to MCS-SVM, while SVM and Sec-SVM (M) typically exhibit lower detection rates. Nevertheless, as these obfuscation attacks do not completely obfuscate the malware code, and the feature changes induced by them are not specifically targeted against any of the given classifiers, the classification performances are not significantly affected. In fact, the DR at 1 percent FPR is never lower than 90 percent. As expected (see Section 4), strategies such as Trivial, String Encryption and Reflection do not affect the system performances significantly, as Drebin only considers system-based API calls, which are not changed by the aforementioned obfuscations. Among these attacks, Class Encryption is the most effective strategy, as it is the only one that more significantly modifies the S5 and S7 feature sets (in particular, the first one), as it can be seen in Fig. 5. Nevertheless, even in this case, as `manifest`-related features are not affected by `DexGuard`-based obfuscations, Drebin still exhibits good detection performances.

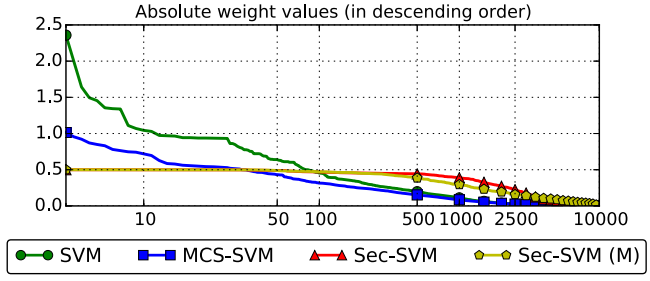

Fig. 3. Absolute weight values in descending order (i.e., $|w_{(1)}| \geq \cdots \geq |w_{(\mathsf{d})}|$), for each classifier (averaged on 10 runs). Flatter curves correspond to more evenly-distributed weights, i.e., more secure classifiers.

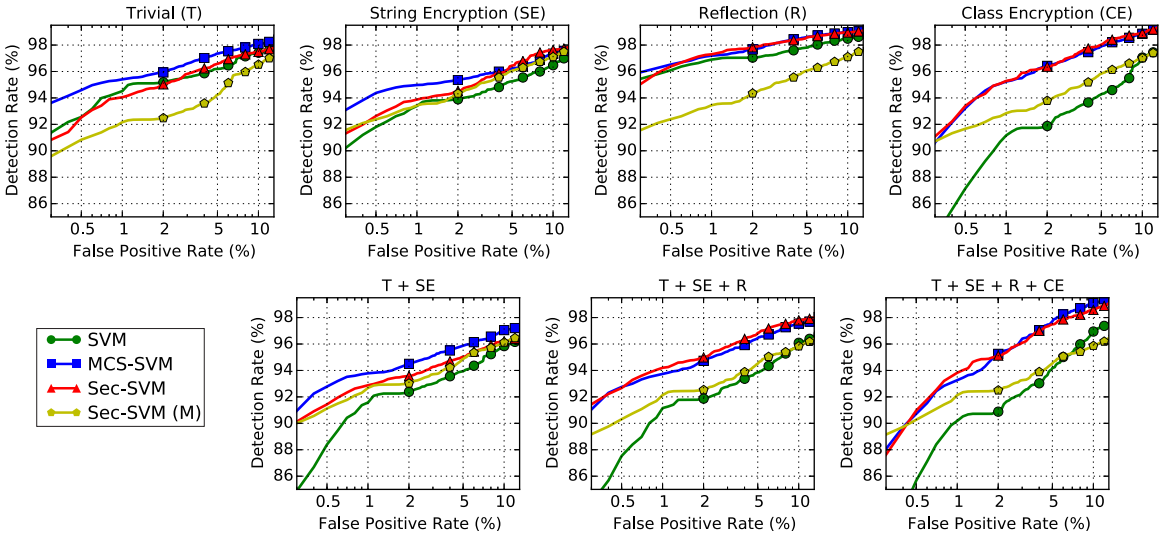

Fig. 4. Mean ROC curves for all classifiers against different obfuscation techniques, computed on the *Contagio* data.

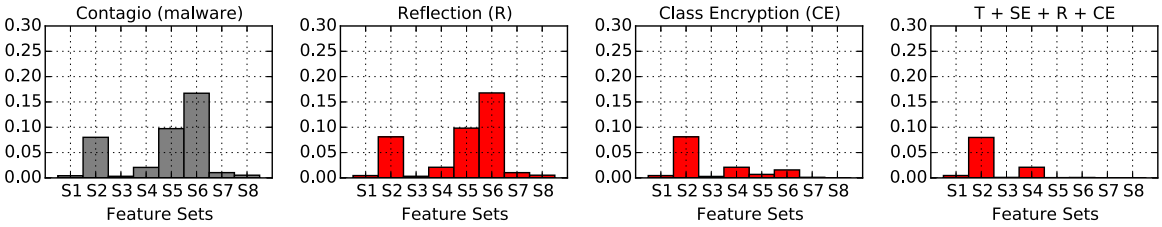

Fig. 5. Fraction of features equal to one in each set (averaged on 10 runs), for non-obfuscated (*leftmost* plot) and obfuscated malware in *Contagio*, with different obfuscation techniques. While obfuscation deletes `dexcode` features (S5-S8), the `manifest` (S1-S4) remains mostly intact.

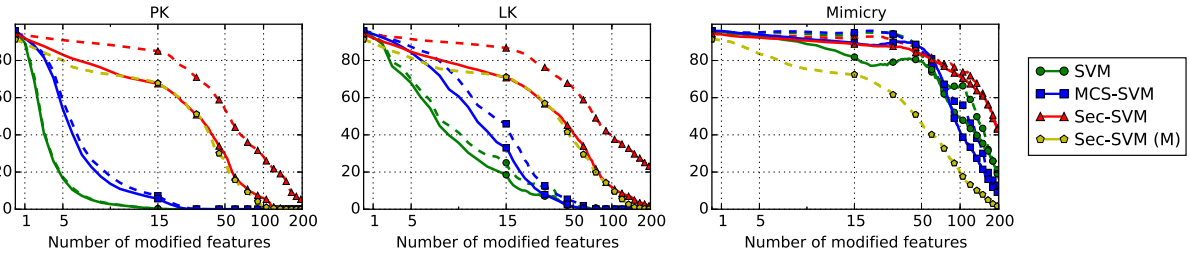

Fig. 6. Detection Rate (DR) at 1 percent False Positive Rate (FPR) for each classifier under the *Perfect-Knowledge* (left), *Limited-Knowledge* (middle), and *Mimicry* (right) attack scenarios, against an increasing number of modified features. Solid (dashed) lines are obtained by simulating attacks with feature addition (feature addition and removal).

Advanced Evasion. We finally report results for the PK, LK, and mimicry attacks in Fig. 6, considering both feature addition, and feature addition and removal. As we are not removing `manifest`-related features, Sec-SVM (M) is clearly tested only against feature-addition attacks. Worth noting, Sec-SVM can drastically improve security compared to the other classifiers, as its performance decreases more gracefully against an increasing number of modified features, especially in the PK and LK attack scenarios. In the PK case, while the DR of Drebin (SVM) drops to 60 percent after modifying only *two* features, the DR of the Sec-SVM decreases to the same amount only when *fifteen* feature values are changed. This means that our Sec-SVM approach can improve classifier security of about *ten* times, in terms of the amount of modifications required to create a malware sample that evades detection. The underlying reason is that Sec-SVM provides more evenly-distributed feature weights, as shown in Fig. 3. Note that Sec-SVM and Sec-SVM (M) exhibit a maximum absolute weight value of 0.5 (on average). This means that, in the worst case, modifying a single feature yields an average decrease of the classification function equal to 0.5, while for MCS-SVM and SVM this decrease is approximately 1 and 2.5, respectively. It is thus clear that, to achieve a comparable decrease of the classification function (i.e., a comparable probability of evading detection), more features should be modified in the former cases. Finally, it is also worth noting that mimicry attacks are less effective, as expected, as they exploit an inferior level of knowledge of the targeted system. Despite this, an interesting insight on the behavior of such attacks is reported in Fig. 7. After modifying a large number of features, the mimicry attack tends to produce a distribution that is very close to that of the benign data (even without removing any `manifest`-related feature). This means that, in terms of their feature vectors, benign and malware samples become very similar. Under these circumstances, no machine-learning technique can separate benign and malware data with satisfying accuracy. The vulnerability of the system may be thus regarded as intrinsic in the choice of the feature representation, rather than in how the classification function is learned.

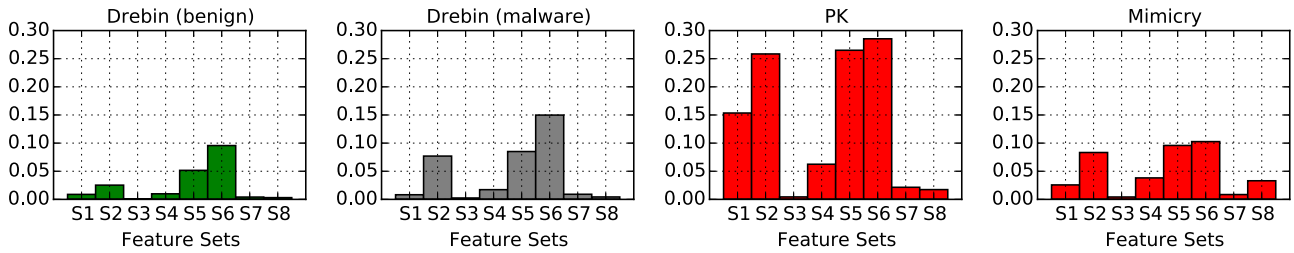

Fig. 7. Fraction of features equal to one in each set (averaged on 10 runs) for benign (*first* plot), non-obfuscated (*second* plot) and DexGuard-based obfuscated malware in *Drebin*, using PK (*third* plot) and mimicry (*fourth* plot) attacks. It is clear that the mimicry attack produces malware samples which are more similar to the benign data than those obtained with the PK attack.

This clearly confirms the importance of designing features that are more difficult to manipulate for an attacker.

Feature Manipulation. To provide some additional insights, in Table 3 we report the top 5 modified features by the PK attack with feature addition and removal for SVM, MCS-SVM, and Sec-SVM. For each classifier, we select the top 5 features by ranking them in descending order of the probability of modification q'. This value is computed as follows. First, the probability q of modifying the kth feature in a malware sample, regardless of the maximum number of admissible modifications, is computed as

$$q = \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x}|y=+1)}\{x_k \neq x'_k\} = p^{\nu}(1-p)^{1-\nu}, \qquad (13)$$

where $\mathbb{E}$ denotes the expectation operator, $p(\boldsymbol{x}|y=+1)$ the distribution of malware samples, x_k and x'_k are the k^{th} feature values before and after manipulation, and p is the probability of observing $x_k = 1$ in malware. Note that $\nu = 1$ if $x_k = 1$, x_k does not belong to the `manifest` sets S1-S4, and the associated weight $\hat{w}_k > 0$, while $\nu = 0$ if $\hat{w}_k < 0$ (otherwise the probability of modification q is zero). This formula denotes compactly that, if a feature can be modified, then it will be changed with probability p (in the case of deletion) or $1-p$ (in the case of insertion). Then, to consider that features associated to the highest absolute weight values are modified more frequently by the attack, with respect to an increasing maximum number m of modifiable features, we compute $q' = \mathbb{E}_m\{q\}$. Considering $m = 1, \ldots, \mathsf{d}$, with uniform probability, each feature will be modified with probability $q' = q\,(\mathsf{d}-r)/\mathsf{d}$, with $r = 0$ for the feature $x_{(1)}$ assigned to the highest absolute weight value, $r = 1$ for the second ranked feature $x_{(2)}$, etc. In general, for the kth-ranked feature $x_{(k)}$, $r = k-1$, for $k = 1, \ldots, \mathsf{d}$. Thus, q' decreases depending on the feature ranking, which in turn depends on the feature weights and the probability p of the feature being present in malware.

Regarding Table 3, note first how the probability of modifying the top features, along with their *relevance* (i.e., their absolute weight value with respect to $\|\boldsymbol{w}\|_1$), decreases from SVM to MCS-SVM, and from MCS-SVM to Sec-SVM. These two observations are clearly connected. The fact that the attack modifies features with a lower probability depends on the fact that weights are more evenly distributed. To better understand this phenomenon, imagine the limit case in which all features are assigned the same absolute weight value. It is clear that, in this case, the attacker could randomly modify any subset of features and obtain the same effect on the classification output; thus, on average, each feature will have the same probability of being modified.

The probability of modifying a feature, however, does not only depend on the weight assigned by the classifier, but also on the probability of being present in malware data, as mentioned before. For instance, if a (non-manifest) feature is present in all malware samples, and it has been assigned a very high positive weight, it will be always removed; conversely, if it only rarely occurs in malware, then it will be deleted only from few samples. This behavior

TABLE 3
Top 5 Modified Features by the PK Evasion Attack with Feature Addition (A) and Removal (R), for SVM, MCS-SVM, and Sec-SVM (Highlighted in Bold)

Feature Set	Feature Name	p	SVM		MCS-SVM		Sec-SVM		A (↑) / R (↓)
S6	susp_calls::android/telephony/gsm/SmsMessage;→getDisplayMessageBody	2.40%	**89.60%**	0.25%	3.99%	0.05%	$<0.03\%$	$<0.02\%$	↑
S1	req_perm::android.permission.USE_CREDENTIALS	0.05%	**65.77%**	0.18%	**67.57%**	0.13%	$<0.03\%$	$<0.02\%$	↑
S1	req_perm::android.permission.WRITE_OWNER_DATA	0.52%	**64.76%**	0.16%	**49.23%**	0.11%	$<0.03\%$	$<0.02\%$	↑
S0	features::android.hardware.touchscreen	0.60%	**64.01%**	0.14%	41.75%	0.09%	$<0.03\%$	$<0.02\%$	↑
S6	susp_calls::android/telephony/gsm/SmsMessage;→getMessageBody	3.50%	**60.13%**	0.13%	17.30%	0.05%	$<0.03\%$	$<0.02\%$	↑
S3	intent_filters::android.intent.action.SENDTO	0.73%	55.70%	0.13%	**60.38%**	0.11%	$<0.03\%$	$<0.02\%$	↑
S6	susp_calls::android/telephony/CellLocation;→requestLocationUpdate	0.05%	50.87%	0.12%	**48.37%**	0.08%	$<0.03\%$	$<0.02\%$	↑
S6	susp_calls::android/net/ConnectivityManager;→getBackgroundDataSetting	0.51%	28.86%	0.07%	**43.59%**	0.09%	$<0.03\%$	$<0.02\%$	↑
S6	susp_calls::android/telephony/TelephonyManager;→getNetworkOperator	46.41%	36.08%	0.17%	43.12%	0.19%	**39.88%**	0.04%	↓
S6	susp_calls::android/net/NetworkInfo;→getExtraInfo	24.81%	19.25%	0.17%	13.42%	0.10%	**10.25%**	0.03%	↓
S6	susp_calls::getSystemService	93.44%	$<3.53\%$	$<0.02\%$	$<0.12\%$	$<0.05\%$	**11.02%**	0.02%	↓
S7	urls::www.searchmobileonline.com	9.42%	4.59%	0.11%	6.91%	0.13%	**4.83%**	0.03%	↓
S6	services::com.apperhand.device.android.AndroidSDKProvider	10.83%	6.78%	0.13%	4.19%	0.09%	**5.14%**	0.03%	↓

The probability of a feature being equal to one in malware data is denoted with p. For each classifier and each feature, we then report two values (averaged on 10 runs): (i) the probability q' that the feature is modified by the attack (left), and (ii) its relevance (right), measured as its absolute weight divided by $\|\boldsymbol{w}\|_1$. If the feature is not modified within the first 200 changes, we report that the corresponding values are only lower than the minimum ones observed. In the last column, we also report whether the feature has been added (↑) or removed (↓) by the attack.

is clearly exhibited by the top features modified by Sec-SVM. In fact, since this classifier basically assigns the same absolute weight value to almost all features, the top modified ones are simply those appearing more frequently in malware. More precisely, in our experiments this classifier, as a result of our parameter optimization procedure, assigns a higher (absolute) weight to features present in malware, and a lower (absolute) weight to features present in benign data (i.e., $|w_k^{ub}| > |w_k^{lb}|$, $k = 1, \ldots, \mathsf{d}$). This is why, conversely to SVM and MCS-SVM, the attack against Sec-SVM tends to remove features, rather then injecting them. To conclude, it is nevertheless worth pointing out that, in general, the most frequently-modified features clearly depend on the data distribution (i.e., on class imbalance, feature correlations, etc.), and not only on the probability of being more frequent in malware. In our analysis, this dependency is intrinsically captured by the dependency of q' on the feature weights learned by the classifier.

Robustness and Regularization. Interestingly, a recent theoretical explanation behind the fact that more features should be manipulated to evade our Sec-SVM can also be found in [46]. In particular, in that work Xu et al. have shown that the *regularized* SVM learning problem, as given in Eq. (7), is equivalent to a non-regularized, *robust* optimization problem, in which the input data is corrupted by a worst-case ℓ_2 (spherical) noise. Note that this noise is *dense*, as it tends to slightly affect all feature values. More generally, Xu et al. [46] have shown that the regularization term depends on the kind of hypothesized noise over the input data. Our evasion attacks are *sparse*, as the attacker aims to minimize the number of modified features, and thus they significantly affect only the most discriminant ones. This amounts to consider an ℓ_1 worst-case noise over the input data. In this case, Xu et al. [46] have shown that the optimal regularizer is the ℓ_∞ norm of $\boldsymbol{w}$. In our Sec-SVM, the key idea is to add a box constraint on $\boldsymbol{w}$, as given in Eq. (8), which is essentially equivalent to consider an additional ℓ_∞ regularizer on $\boldsymbol{w}$, consistently with the findings in [46].

7 Limitations and Open Issues

Despite the very promising results achieved by our Sec-SVM, it is clear that such an approach exhibits some intrinsic limitations. First, as Drebin performs a static code analysis, it is clear that also Sec-SVM can be defeated by more sophisticated encryption and obfuscation attacks. However, it is also worth remarking that this is not a vulnerability of the learning algorithm itself, but rather of the chosen feature representation, and for this reason we have not considered these attacks in our work. A similar behavior is observed when a large number of features is modified by our evasion attacks, and especially in the case of mimicry attacks (see Section 6), in which the manipulated malware samples almost exactly replicate benign data (in terms of their feature vectors). This is again possible due to an intrinsic vulnerability of the feature representation, and no learning algorithm can clearly separate such data with satisfying accuracy. Nevertheless, this problem only occurs when malware samples are significantly modified and, as pointed out in Section 3.6, it might be very difficult for the attacker to do that without compromising their intrusive functionality, or without leaving significant traces of adversarial manipulation. For example, the introduction of changes such as reflective calls requires a careful manipulation of the Dalvik registers (e.g., verifying that old ones are correctly re-used and that new ones can be safely employed). A single mistake in the process can lead to verification errors, and the application might not be usable anymore (we refer the reader to [23], [24] for further details). Another limitation of our approach may be its unsatisfying performance under PK and LK attacks, but this can be clearly mitigated with simple countermeasures to prevent that the attacker gains sufficient knowledge of the attacked system, such as frequent system re-training and diversification of training data collection [9]. To summarize, although our approach is clearly not bulletproof, we believe that it significantly improves the security of the baseline Drebin system (and of the standard SVM algorithm).

8 Conclusions and Future Work

Recent results in the field of adversarial machine learning and computer security have confirmed the intuition pointed out by Barreno et al. [4], [5], [25], namely, that machine learning itself can introduce specific vulnerabilities in a security system, potentially compromising the overall system security. The underlying reason is that machine-learning techniques have not been originally designed to deal with intelligent and adaptive attackers, who can modify their behavior to mislead the learning and classification algorithms.

The goal of this work has been, instead, to show that machine learning *can* be used to improve system security, if one follows an *adversary-aware* approach that proactively anticipates the attacker. To this end, we have first exploited a general framework for assessing the security of learning-based malware detectors, by modeling attackers with different goals, knowledge of the system, and capabilities of manipulating the data. We have then considered a specific case study involving Drebin, an Android malware detection tool, and shown that the performance of Drebin can be significantly downgraded in the presence of skilled attackers that can carefully manipulate malware samples to evade classifier detection. The main contribution of this work has been to define a novel, theoretically-sound learning algorithm to train linear classifiers with more evenly-distributed feature weights. This approach allows one to improve system security (in terms of requiring a much higher number of careful manipulations to the malware samples), without significantly affecting computational efficiency.

A future development of our work, which may further improve classifier security, is to extend our approach for secure learning to nonlinear classifiers, e.g., using *nonlinear kernel functions*. Although *nonlinear kernels* can not be directly used in our approach (due to the presence of a linear constraint on $\boldsymbol{w}$), one may exploit a trick known as the *empirical kernel mapping*. It consists of first mapping samples onto an *explicit* (approximate) kernel space, and then learning a linear classifier on that space [39]. We would like to remark here that also investigating the trade-off between *sparsity* and *security* highlighted in Section 5.2 may provide interesting insights for future work. In this respect, the recent findings in [46] related to robustness and regularization of learning algorithms (briefly summarized at the end of Section 6) may provide inspiring research directions.

Another interesting future extension of our approach may be to explicitly consider, for each feature, a different level of robustness against the corresponding adversarial manipulations. In practice, however, the *agnostic* choice of assuming equal robustness for all features may be preferred, as it may be very difficult to identify features that are more difficult to manipulate. If categorizing features according to their robustness to adversarial manipulations is deemed feasible, instead, then this knowledge may be incorporated into the learning algorithm, such that higher (absolute) weight values are assigned to more robust features.

It is finally worth remarking that we have also recently exploited the proposed learning algorithm to improve the security of PDF and Javascript malware detection systems against *sparse* evasion attacks [20], [38]. This witnesses that our proposal does not only provide a first, concrete example of how machine learning can be exploited to improve *security* of Android malware detectors, but also of how our design methodology can be readily applied to other learning-based malware detection tasks.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," in *Proc. Int. Conf. Secur. Privacy Commun. Netw.*, 2013, pp. 86–103.

[2] I. Arce, "The weakest link revisited [information Sec.]," *IEEE Secur. Privacy*, vol. 1, no. 2, pp. 72–76, Mar. 2003.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proc. 21st NDSS*, 2014, p. 12.

[4] M. Barreno, B. Nelson, A. Joseph, and J. Tygar, "The security of machine learning," *Mach. Learn.*, vol. 81, pp. 121–148, 2010.

[5] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar, "Can machine learning be secure?" in *Proc. ACM Symp. Inf. Comput. Commun. Secur.*, 2006, pp. 16–25.

[6] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 73–84.

[7] B. Biggio, et al., "Evasion attacks against machine learning at test time," in *Proc. Eur. Conf. Mach. Learn. Principles Practice Knowl. Discovery Databases*, 2013, pp. 387–402.

[8] B. Biggio, G. Fumera, and F. Roli, "Multiple classifier systems for robust classifier design in adversarial environments," *Int. J. Mach. Learn. Cybern.*, vol. 1, no. 1, pp. 27–41, 2010.

[9] B. Biggio, G. Fumera, and F. Roli, "Pattern recognition systems under attack: Design issues and research challenges," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 28, no. 7, 2014, Art. no. 1460002.

[10] B. Biggio, G. Fumera, and F. Roli, "Security evaluation of pattern classifiers under attack," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 4, pp. 984–996, Apr. 2014.

[11] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proc. 29th Int. Conf. Mach. Learn.*, 2012, pp. 1807–1814.

[12] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. 21st Int. Conf. Comput. Statist.*, 2010, pp. 177–186.

[13] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[14] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.

[15] M. Brückner, C. Kanzow, and T. Scheffer, "Static prediction games for adversarial learning problems," *J. Mach. Learn. Res.*, vol. 13, pp. 2617–2654, Sep. 2012.

[16] V. Cherkassky and Y. Ma, "Another look at statistical learning theory and regularization," *Neural Netw.*, vol. 22, no. 7, pp. 958–969, 2009.

[17] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. 148, Jul. 1997.

[18] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, 1995.

[19] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, "Adversarial classification," in *Proc. 10th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2004, pp. 99–108.

[20] A. Demontis, P. Russu, B. Biggio, G. Fumera, and F. Roli, "On Sec. and sparsity of linear classifiers for adversarial settings," in *Proc. Joint IAPR Int. Workshop Structural Syntactic Statist. Pattern Recognit.*, 2016, pp. 322–332.

[21] A. Globerson and S. T. Roweis, "Nightmare at test time: Robust learning by feature deletion," in *Proc. 23rd Int. Conf. Mach. Learn.*, 2006, pp. 353–360.

[22] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, no. 8, pp. 832–844, Aug. 1998.

[23] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz, "Evaluating analysis tools for android apps: Status quo and robustness against obfuscation," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 139–141.

[24] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz, "Evaluating analysis tools for android apps: Status quo and robustness against obfuscation," Horst Görtz Institute for IT Sec., Bochum, Germany, Tech. Rep. TR-HGI-2016–003, 2016.

[25] L. Huang, A. D. Joseph, B. Nelson, B. Rubinstein, and J. D. Tygar, "Adversarial Machine learning," in *Proc. 4th ACM Workshop Artif. Intell. Secur.*, 2011, pp. 43–57.

[26] A. Kolcz and C. H. Teo, "Feature weighting for improved classifier robustness," in *Proc. 6th Conf. Email Anti-Spam*, 2009, p. 8.

[27] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. 39th Annu. Int. Comput. Softw. Appl. Conf.*, 2015, pp. 422–433.

[28] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "ANDRUBIS—1,000,000 apps later: A view on current android malware behaviors," in *Proc. 3rd Int. Workshop Building Anal. Datasets Gathering Experience Returns Secur.*, 2014, pp. 3–17.

[29] D. Lowd and C. Meek, "Adversarial learning," in *Proc. 11th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2005, pp. 641–647.

[30] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks," *Comput. Secur.*, vol. 51, pp. 16–31, Jun. 2015.

[31] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf.*, 2007, pp. 421–430.

[32] J. Newsome, B. Karp, and D. Song, "Paragraph: Thwarting signature learning by training maliciously," in *Proc. 9th Int. Conf. Recent Advances Intrusion Detection*, 2006, pp. 81–105.

[33] H. Peng, et al., "Using probabilistic generative models for ranking risks of android apps," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 241–252.

[34] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proc. IEEE Symp. Privacy*, 2006, pp. 17–31.

[35] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, *Uranine: Real-Time Privacy Leakage Monitoring Without System Modification for Android*. Cham, Switzerland: Springer, 2015, pp. 256–276.

[36] B. Reaves, et al., "*droid: Assessment and evaluation of android application analysis tools," *ACM Comput. Surveys*, vol. 49, no. 3, pp. 55:1–55:30, Oct. 2016.

[37] S. Roy, et al., "Experimental study with real-world data for android app Security analysis using machine learning," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2016, pp. 81–90.

[38] P. Russu, A. Demontis, B. Biggio, G. Fumera, and F. Roli, "Secure kernel machines against evasion attacks," in *Proc. 9th ACM Workshop Artif. Intell. Secur.*, 2016, pp. 59–69.

[39] B. Schölkopf, et al., "Input space versus feature space in kernel-based methods," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 1000–1017, Sep. 1999.

[40] C. H. Teo, A. Globerson, S. Roweis, and A. Smola, "Convex learning with invariances," in *Proc. 20th Int. Conf. Neural Inf. Process. Syst.*, 2008, pp. 1489–1496.

[41] V. N. Vapnik, *The Nature of Statistical Learning Theory*. Berlin, Germany: Springer, 1995.

[42] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2008, p. 16.

[43] N. Šrndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *Proc. IEEE Symp. Privacy*, 2014, pp. 197–211.

[44] G. Wang, T. Wang, H. Zheng, and B. Y. Zhao, "Man versus machine: Practical adversarial detection of malicious crowdsourcing workers," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 239–254.
[45] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli, "Is feature selection secure against training data poisoning?" in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, 2015, vol. 37, pp. 1689–1698.
[46] H. Xu, C. Caramanis, and S. Mannor, "Robustness and regularization of support vector Machines," *J. Mach. Learn. Res.*, vol. 10, pp. 1485–1510, Jul. 2009.
[47] W. Yang, et al., *AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware*. Cham, Switzerland: Springer, 2015, pp. 359–381.
[48] F. Zhang, P. Chan, B. Biggio, D. Yeung, and F. Roli, "Adversarial feature selection against evasion attacks," *IEEE Trans. Cybern.*, vol. 46, no. 3, pp. 766–777, Mar. 2016.
[49] T. Zhang, "Solving large scale linear prediction problems using SGD algorithms," in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, pp. 116–123.
[50] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Privacy*, 2012, pp. 95–109.

Ambra Demontis (S'16) received the MSc (with Hons.) degree in information technology from the University of Cagliari, Italy, in 2014. She is now working toward the PhD degree in the Department of Electrical and Electronic Engineering, University of Cagliari. Her current research interests include machine learning, computer security, and biometrics. She is a student member of the IEEE and the IAPR.

Marco Melis (S'16) received the BSc degree in electronic engineering from the University of Cagliari, Italy, in February 2014. Since 2014, he has been in the Department of Electrical and Electronic Engineering, University of Cagliari. His research activity is focused on machine learning, kernel methods, and biometric recognition systems. He is a student member of the IEEE.

Battista Biggio (SM'17) received the MSc (Hons.) degree in electronic engineering and the PhD degree in electronic engineering and computer science from the University of Cagliari, Italy, in 2006 and 2010, respectively. Since 2007, he has been in the Department of Electrical and Electronic Engineering, University of Cagliari, where he is currently an assistant professor. In 2011, he visited the University of Tübingen, Germany, and worked on the security of machine learning to training data poisoning. His research interests include secure machine learning, multiple classifier systems, kernel methods, biometrics, and computer security. He serves as a reviewer for several international conferences and journals. He is a senior member of the IEEE and a member of the IAPR.

Davide Maiorca (M'16) received the MSc (Hons.) degree in electronic engineering and the PhD degree in electronic and computer engineering from the University of Cagliari, Italy, in 2012 and 2016, respectively. In 2013, he visited the Systems Security Group, Ruhr-Universität Bochum, guided by Prof. Dr. Thorsten Holz, and worked on advanced obfuscation of Android malware. He now holds a post-doctoral position with the University of Cagliari. His current research interests include adversarial machine learning, malware in documents and Flash applications, Android malware and mobile fingerprinting. He has been a member of the 2016 IEEE Security & Privacy Student Program Committee. He is a member of the IEEE.

Daniel Arp received the diploma degree in computer engineering from Technische Universität Berlin, in 2012 and worked as a research assistant with the University of Göttingen subsequently. He is working toward the PhD degree working in the Computer Security Group, Technische Universität Braunschweig. His research focuses on the application of learning methods for malware detection and privacy protection.

Konrad Rieck received the graduate degree from Freie Universität Berlin, in 2004 and the doctorate in computer science from Technische Universität Berlin, in 2009. He is a professor with Technische Universität Braunschweig, where he leads the Institute of System Security. Prior to taking this position, he has been working with the University of Göttingen, Technische Universität Berlin, and Fraunhofer Institute FIRST. He received the CAST/GI Dissertation Award IT-Security and a Google Faculty Research Award. His interests revolve around computer security and machine learning, including the detection of computer attacks, the analysis of malicious code, and the discovery of vulnerabilities.

Igino Corona received the MSc degree in electronic engineering and the PhD in computer engineering from the University of Cagliari, Italy, in 2006 and 2010, respectively. In 2009, he also worked in the Georgia Tech Information Security Center as a research scholar. He is currently a post-doctoral researcher with the University of Cagliari. His main research interests include web security, detection of fast flux networks, adversarial machine learning, and web intrusion detection.

Giorgio Giacinto (SM'10) received the MS degree in electrical engineering, in 1994 and the PhD in computer engineering, in 1999. He is an associate professor of computer engineering with the University of Cagliari, Italy. His research interests include pattern recognition, computer security, image classification and retrieval. He has been coordinator of several international research projects in pattern recognition and computer security. Since 2012, he has been the organizer of the Summer School on Computer Security and Privacy "Building Trust in the Information Age". He is a senior member of the ACM and the IEEE.

Fabio Roli (F'12) received the PhD degree in electronic engineering from the University of Genoa, Italy. He was a research group member of the University of Genoa (88-94). He was adjunct professor with the University of Trento ('93-'94). In 1995, he joined the Department of Electrical and Electronic Engineering, University of Cagliari, where he is now professor of computer engineering and head of the Research Laboratory on Pattern Recognition and Applications. His research activity is focused on the design of pattern recognition systems and their applications. He was a very active organizer of international conferences and workshops, and established the popular workshop series on multiple classifier systems. He is a fellow of the IEEE and the IAPR.