SecureDroid: Enhancing Security of Machine Learning-based Detection against Adversarial Android Malware Attacks

Lingwei Chen, Shifu Hou, Yanfang Ye ✉
Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV, USA
{lgchen,shhou}@mix.wvu.edu,yanfang.ye@mail.wvu.edu

ABSTRACT

With smart phones being indispensable in people's everyday life, Android malware has posed serious threats to their security, making its detection of utmost concern. To protect legitimate users from the evolving Android malware attacks, machine learning-based systems have been successfully deployed and offer unparalleled flexibility in automatic Android malware detection. In these systems, based on different feature representations, various kinds of classifiers are constructed to detect Android malware. Unfortunately, as classifiers become more widely deployed, the incentive for defeating them increases. In this paper, we explore the security of machine learning in Android malware detection on the basis of a learning-based classifier with the input of a set of features extracted from the Android applications (apps). We consider different importances of the features associated with their contributions to the classification problem as well as their manipulation costs, and present a novel feature selection method (named *SecCLS*) to make the classifier harder to be evaded. To improve the system security while not compromising the detection accuracy, we further propose an ensemble learning approach (named *SecENS*) by aggregating the individual classifiers that are constructed using our proposed feature selection method *SecCLS*. Accordingly, we develop a system called ***SecureDroid*** which integrates our proposed methods (i.e., *SecCLS* and *SecENS*) to enhance security of machine learning-based Android malware detection. Comprehensive experiments on the real sample collections from Comodo Cloud Security Center are conducted to validate the effectiveness of *SecureDroid* against adversarial Android malware attacks by comparisons with other alternative defense methods. Our proposed secure-learning paradigm can also be readily applied to other malware detection tasks.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Machine learning algorithms**;

KEYWORDS

Machine Learning, Adversarial Attack, Android Malware Detection

ACSAC 2017, December 4–8, 2017, Orlando, FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134636

1 INTRODUCTION

Smart phones have become increasingly ubiquitous in people's everyday life, which are used to perform tasks like social networking, financial management, and entertainment. In recent years, there has been an exponential growth in the number of smart phone users around the world: it is estimated that the worldwide smart phone market will reach a total of 1.53 billion units in 2017, and 1.77 billion units in 2021 [31]. Designed as an open, free, and programmable operation system, Android as one of the most popular smart phone platforms has dominated the current market share [29]. However, the openness of Android not only attracts the developers for producing legitimate apps, but also attackers to disseminate malware (short for ***ma***licious soft***ware***) onto unsuspecting users to disrupt the mobile operations. Today, a lot of Android malware (e.g., HummingWhale, BankBot, Geinimi, DriodKungfu, RootSmart, and Obad) is released on the markets [32, 39]. According to Symantec's Internet Security Threat Report [35], one in every five Android apps were actually malware. This has posed serious threats to the smart phone users, such as stealing user's credentials, auto-dialing premium numbers, and sending SMS messages without user's permission [13]. As a result, the detection of Android malware is of major concern to both the anti-malware industry and researchers.

In order to combat the evolving Android malware attacks, systems applying machine learning techniques have been developed for automatic Android malware detection in recent years [20–22, 36, 37, 40, 45]. In these systems, based on different feature representations (e.g., system call graphs [20], dynamic behaviors [37], or Application Programming Interface (API) call blocks [21]), various kinds of classification approaches, such as Support Vector Machine [47], Random Forest [1] and Deep Neural Network [20, 21], are used for model construction to detect malicious apps. Though these techniques offer unparalleled flexibility in automatic Android malware detection, machine learning itself may open the possibility for an adversary who maliciously "mis-trains" a classifier (e.g., by changing data distribution or feature importance) in the detection system. When the learning system is deployed in a real-world environment, it is of a great interest for the attackers to actively manipulate the data to make the classifier produce minimum true positive (i.e., maximumly misclassifying malware as benign), using some combination of prior knowledge, observation, and experimentation.

Defenders and attackers are always engaged in a never-ending arms race. At each round, both of them try to analyze the methodologies and vulnerabilities of each other, and develop their own optimal strategies to overcome the opponents [6, 9], which has led to considerable countermeasures of variability and sophistication between them. For example, Android malware attackers

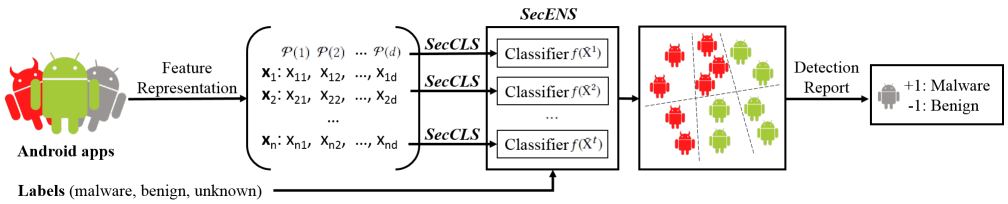


Figure 1: An overview of system architecture of *SecureDroid*. In the system, the collected apps are first represented as d-dimensional binary feature vectors. Then *SecCLS* is applied to select a set of features (each feature i is selected with probability $\mathcal{P}(i)$) to construct a more secure classifier. *SecENS* is later exploited to aggregate different individual classifiers built using *SecCLS* to classify malicious and benign apps. For a new app, based on the extracted features, it will be predicted as either malicious or benign based on the trained classification model.

employ techniques such as repackaging and obfuscation to bypass the signature-based detection and defeat attempts to analyze their inner mechanisms [20]. Currently, the issues of understanding machine learning security in adversarial settings are starting to be leveraged [10, 11, 15, 25, 27, 30, 34, 46], from either adversarial or defensive perspectives. However, the application of adversarial machine learning into Android malware detection domain has been scarce. With the popularity of machine learning based detections, such adversaries will sooner or later present [42].

In this paper, we investigate the adversarial Android malware attacks and aim to enhance security of machine learning-based detection against such attacks. In the adversarial point of view, to conduct a practical attack, attackers intend to find the features which are easy to be manipulated (i.e., features with low costs being manipulated) and minimize the manipulations (i.e., modify the features as less as possible) to bypass the detection. For example, to evade the detection, attackers may manipulate the Android Trojan *"net.Mwkek"* by injecting the permission of *"BATTERY_STATS"* which is frequently used in benign apps in the *manifest* file instead of removing suspicious permission of "*SEND_SMS*", since feature addition is usually cost-effective and safer than feature elimination to bypass the detection while preserves the semantics and intrusive functionality of the original malicious app. In contrast, to be resilient against the adversarial attacks, an ideal defense should make the attackers cost-expensive and maximize their manipulations to evade the detection. In this paper, resting on the analysis of a set of features (i.e., permissions, filtered intents, API calls, and new-instances) extracted from the Android apps, we explore the security of machine learning in Android malware detection on the basis of a learning-based classifier. To make the classifier harder to be evaded, we first present a novel feature selection method (named *SecCLS*) to build the classifier, by taking consideration of different importances of the features associated with their contributions to the classification problem as well as their manipulation costs. To improve the system security while not compromising the detection accuracy, we further propose an ensemble learning approach (named *SecENS*) by aggregating the individual classifiers that are constructed using the proposed feature selection method *SecCLS*. Accordingly, we develop a system called ***SecureDroid*** which integrates both *SecCLS* and *SecENS* to secure machine learning-based Android malware detection. The system architecture of *SecureDroid* is shown in Figure 1, which has the following major traits:

- ***Novel feature selection method for more secure classifier construction:*** For attackers, the importance of a feature in the adversarial settings depends on: (i) its cost being manipulated, which is determined by the type of feature (e.g., permission vs. API call) and its manipulation method (e.g., feature addition vs. elimination), and (ii) its contribution to the classification problem, which is weighted by the learning system based on the training data. We thoroughly assess the adversary behaviors and present a novel feature selection method (named *SecCLS*) to build more secure classifier by enforcing attackers to increase the evasion costs and maximize the manipulations.
- ***An ensemble learning approach to improve system security while not compromising detection accuracy:*** To aggregate different individual classifiers constructed using our proposed feature selection method *SecCLS*, we introduce an ensemble learning approach (named *SecENS*) against the adversarial attacks in Android malware detection. In the ensemble framework, we not only consider the diversity of individual classifiers but also the integration of the whole feature space.
- ***A practical and resilient system against adversarial Android malware attacks:*** We collect two sample sets (including 8,046 apps and 72,891 apps respectively) from Comodo Cloud Security Center. Based on these real sample collections, we develop a resilient system *SecureDroid* which integrates our proposed methods to enhance the security of machine learning model in Android malware detection. A series of comprehensive experiments are conducted and the results demonstrate that *SecureDroid* can bring the detection system back up to the desired performance level against different kinds of adversarial attacks, including brute-force attacks, anonymous attacks and well-crafted attacks.

The rest of the paper is organized as follows. Section 2 defines the problem of machine learning-based Android malware detection. Section 3 discusses the adversarial Android malware attacks. Section 4 introduces our proposed methods in detail. Based on the real sample collections from Comodo Cloud Security Center, Section 5 systematically evaluates the effectiveness of our developed system *SecureDroid* which integrates the proposed methods against different kinds of adversarial attacks, by comparisons with other alternative defense methods. Section 6 discusses the related work. Finally, Section 7 concludes.

2 MACHINE LEARNING-BASED ANDROID MALWARE DETECTION

An Android malware detection system using machine learning techniques attempts to identify variants of known malware or zero-day malware through building a classification model based on the labeled training samples and predefined feature representations. In this section, we introduce a learning-based classifier based on the feature representations of Android apps with preliminaries.

2.1 Preliminaries

Unlike traditional desktop based Portable Executable (PE) file, Android app is compiled and packaged in a single archive file (with an .apk suffix) that contains the manifest file, Dalvid executable (dex) file, resources, and assets.

Manifest file. Android defines a component-based framework for developing mobile apps, which is composed of four different types of components [20]: *Activities* provide Graphical User Interface (GUI) functionality to enable user interactivity; *Services* are background communication processes that pass messages between the components of the app and communicate with other apps; *Broadcast Receivers* are background processes that respond to system-wide broadcast messages as necessary; and *Content Providers* act as database management systems that manage the app data. Android app must declare its components in the manifest file which retains information about its structure. Before the Android system can start an app component, the system must know that the component exists by reading the app's manifest file. The manifest file actually works as a road map to ensure that each app can function properly in the Android system. The actions of each component are further specified through *filtered intents* which declare the types of intents that an activity, service, or broadcast receiver can respond to [2]. For example, through filtered intents, an activity can initiate a phone call or a broadcast receiver can monitor SMS message. The manifest file also contains a list of *permissions* requested by the app to perform functions (e.g., access Internet). Since permissions and filtered intents can reflect the interaction between an app and other apps or operation system, we extract them from manifest file as features to represent Android apps.

Dalvid executable (dex). Android apps are usually developed with Java. Development environments (e.g., Eclipse) convert the Java source codes into Dalvik executable (dex) files which can be run on the Dalvik Virtual Machine (DalvikVM)[1] in Android. Dex is a file format that contains compiled code written for Android and can be interpreted by the DalvikVM, which includes all the user-implemented methods and classes. Dex file always contains *API calls* that are used by the Android apps in order to access operating system functionality and resources, and *new-instances* which can be used to create new instances of classes from operating system classes. Therefore, both API calls and new-instances in the dex file can be used to represent the behaviors of an Android app. To extract them from a dex file, since dex file is unreadable, we (1) first use the reverse engineering tool APKTool[2] to decompile the dex file into smali code (i.e., the intermediate but interpreted code between Java and DalvikVM); and (2) then parse the converted smali code to extract these two kinds of features.

In this paper, we perform static analysis on the collected Android apps and extract the above features (i.e., permissions and filtered intents from manifest file, API calls and new-instances from dex file) to represent the apps. Though static analysis has unequivocal limitations, since it is not feasible to analyze malicious code that is thoroughly obfuscated or decrypted at runtime. For this reason, considering such attacks would be irrelevant for the scope of our work. Our focus is rather to understand and to enhance the security properties of learning-based system against a wide class of adversarial attacks. The above features are exploited as a case study which facilitate the understanding of our further proposed approach, while other feature extractions are also applicable in our further investigation.

2.2 Feature Representation

To represent each collected Android app, we first extract the features and convert them into a vector space, so that it can be fed to the classifier either for training or testing. As described in Section 2.1, for the collected apps, we extract four sets of features (***S1 – S4***) to represent them (shown in Table 1): permissions (*S1*) and filtered intents (*S2*) from manifest files, API calls (*S3*) and new-instances (*S4*) from dex files.

Table 1: Illustration of extracted features

	Features	Examples
Manifest	*S1:* Permissions	*READ_PHONE_STATE* *INTERNET*
	S2: Filtered Intents	*intent.action.MAIN* *vending.INSALL_REFERER*
Dex	*S3:* API calls	*getSimSerialNumber* *containsHeader*
	S4: New-Instances	*Ljave/util/HashMap* *Landroid/app/ProgressDialog*

Resting on the above extracted features, we denote our dataset D to be of the form $D = \{\mathbf{x}_i, y_i\}_{i=1}^{n}$ of n apps, where $\mathbf{x}_i$ is the features extracted from app i, and y_i is the class label of app i ($y_i \in \{+1, -1, 0\}$, +1: malicious, −1: benign, and 0: unknown). Let d be the number of all extracted features in *S1 – S4* in dataset D. Each app can then be represented by a binary feature vector:

$$\mathbf{x}_i = \begin{pmatrix} 0 \\ \dots \\ 1 \\ 1 \\ \dots \\ 0 \\ 1 \\ \dots \\ 0 \\ 1 \\ \dots \\ 1 \end{pmatrix} \rightarrow \begin{matrix} \text{READ\_PHONE\_STATE} \\ \dots \\ \text{INTERNET} \\ \text{intent.action.MAIN} \\ \dots \\ \text{vending.INSALL\_REFERER} \\ \text{getSimSerialNumber} \\ \dots \\ \text{containsHeader} \\ \text{Ljave/util/HashMap} \\ \dots \\ \text{Landroid/app/ProgressDialog} \end{matrix} \begin{matrix} \} \textit{S1:} \text{ Permissions} \\ \\ \\ \} \textit{S2:} \text{ Filtered Intents} \\ \\ \\ \} \textit{S3:} \text{ API calls} \\ \\ \\ \} \textit{S4:} \text{ New-Instances} \end{matrix}$$

[1] https://source.android.com/devices/tech/dalvik/.
[2] http://ibotpeaches.github.io/Apktool/

where $\mathbf{x}_i \in \mathbb{R}^d$, and $x_{ij} = \{0, 1\}$ (i.e., if app i includes feature j, then $x_{ij} = 1$; otherwise, $x_{ij} = 0$).

2.3 Learning-based Classifier for Android Malware Detection

The problem of machine learning based Android malware detection can be stated in the form of: $f : \mathcal{X} \to \mathcal{Y}$ which assigns a label $y \in \mathcal{Y}$ (i.e., -1 or $+1$) to an input app $\mathbf{x} \in \mathcal{X}$ through the learning function f. A general linear classification model for Android malware detection can be thereby denoted as:

learning function

$$\mathbf{f} = \text{sign}(f(\mathbf{X})) = \text{sign}(\mathbf{X}^T\mathbf{w} + \mathbf{b}), \quad (1)$$

where $\mathbf{f}$ is a vector, each of whose elements is the label (i.e., malicious or benign) of an app to be predicted, each column of matrix $\mathbf{X}$ is the feature vector of an app, $\mathbf{w}$ is the weight vector and $\mathbf{b}$ is the biases. More specifically, a machine learning system on the basis of a linear classifier can be formalized as an optimization problem [44]:

线性分类模型

$$\underset{\mathbf{f},\mathbf{w},\mathbf{b};\xi}{\text{argmin}} \frac{1}{2}||\mathbf{y} - \mathbf{f}||^2 + \frac{1}{2\beta}\mathbf{w}^T\mathbf{w} + \frac{1}{2\gamma}\mathbf{b}^T\mathbf{b} + \xi^T(\mathbf{f} - \mathbf{X}^T\mathbf{w} - \mathbf{b}), \quad (2)$$

subject to Eq. (1), where $\mathbf{y}$ is the labeled information vector, ξ is Lagrange multiplier which is a strategy for finding the local minima of $\frac{1}{2}||\mathbf{y}-\mathbf{f}||^2$ subject to $\mathbf{f}-\mathbf{X}^T\mathbf{w}-\mathbf{b} = 0$, β and γ are the regularization parameters. Note that Eq. (2) is a general linear classifier (denoted as *Original-Classifier* throughout the paper) consisting of specific loss function and regularization terms. Without loss of generality, the equation can be transformed into different linear models depending on the choices of loss function and regularization terms, such as Support Vector Machine (SVM).

3 ADVERSARIAL ATTACKS

In Android malware detection, a learning-based system is to detect malicious apps based on the trained classification model and prevent them from interfering users' smart phones. On the contrast, attackers would like to violate the security context by either (a) allowing malicious apps to be misclassified as benign (*integrity attack*) or (b) creating a denial of service in which benign apps are incorrectly classified as malicious (*availability attack*) [3, 4]. In this paper, we focus on the integrity attack, also called adversarial attack.

分为两类：integerity availability

Adversarial attacks can generally be modeled as an optimization problem: given an original malicious app $\mathbf{x} \in \mathcal{X}^+$, the adversarial attacks attempt to manipulate its features to be detected as benign (i.e., $\mathbf{x}' \in \mathcal{X}^-$), with the minimal evasion cost. In this section, we present how attackers can achieve such attacks.

考虑成本要素

3.1 Feature Manipulation

To conduct an adversarial attack, attackers would manipulate the features of a malicious app to evade the detection. As described in Section 2.2, given an Android app, after feature extraction, it can be represented by a binary feature vector. Then a typical manipulation can be either adding or eliminating a binary in the vector.

- **Feature Addition.** In this scenario, attackers can autonomously inject a feature in the app (i.e., set 0 to 1). For example, they can add permissions in the manifest file without influence on other existing functionalities; they can also inject API calls in a dead code or methods which will be never called by any invoke instructions in the dex file.
- **Feature Elimination.** In this setting, attackers may hide or remove a feature from the app (i.e., set 1 to 0) while not affecting the intrusive functionality they want to execute. For example, attackers can hide the information stored as strings by encryption and decrypting it at runtime.

Either feature addition or elimination, both settings should retain the semantics and intrusive functionality of the original app after manipulations. In such case, feature addition is easier and safer when the injection is not directly executed by the app (as examples shown above). However, if attackers want to inject a suspicious API call to the dex file being executed by the app, it will be more sophisticated and may influence the semantics of the app. Feature elimination is usually more complicated, such as, removing permissions from the manifest file is not always practical since it may limit the functionalities of the app. Therefore, conducting an adversarial attack that needs to manipulate a lot of features while not compromising the malicious functionalities may not always be feasible. In this respect, attackers may need to implement a well-crafted attack by taking consideration of the evasion cost.

well-crafted

3.2 Evasion Cost

The evasion cost can be decided by the number of binaries that are changed from $\mathbf{x}$ to $\mathbf{x}'$ by attackers, which is denoted as

$$C(\mathbf{x}', \mathbf{x}) = ||\mathbf{c}^T(\mathbf{x}' - \mathbf{x})||_p^p, \quad (3)$$

where $\mathbf{c}$ is a vector whose element denotes the corresponding cost of changing a feature, and p is a real number. The evasion cost function can be considered as ℓ_1-norm or ℓ_2-norm depending on the feature space. For attackers, the manipulation cost c_i for each feature is different. For example, the cost of removing a permission from the manifest file is much higher than injecting an API call in a dead code in the dex file. Therefore, the manipulation cost c_i for each feature is practically significant, which is determined by the feature type (e.g., permission vs. API call) and manipulation method (e.g., addition vs. elimination). Furthermore, for the reasons aforementioned, it's impractical for attackers to modify a malicious app into benign at any cost (i.e., manipulating a large number of features). Thus, there is an upper limit of the maximum manipulations that can be made to the original malicious app $\mathbf{x}$. That is, the manipulation function $\mathcal{A}(\mathbf{x})$ can be formulated as

$$\mathcal{A}(\mathbf{x}) = \begin{cases} \mathbf{x}' & \text{sign}(f(\mathbf{x}')) = -1 \textit{ and } C(\mathbf{x}', \mathbf{x}) \le \delta_{\max} \\ \mathbf{x} & \text{otherwise} \end{cases}, \quad (4)$$

where the malicious app is manipulated to be misclassified as benign only if the evasion cost is less than or equal to a maximum cost $\delta_{\max}$.

3.3 Attack Model

In practice, though attackers may know differently about the targeted learning system [33], they always have the following two competing objectives: (1) maximize the number of malicious apps being classified as benign, and (2) minimize the evasion cost for optimal attacks over the learning-based classifier [25]. Specifically,

找一个变化后的样本x'，它“尽量”被分类为良性，且变换成本最小。

the adversarial attack model can be formulated as:

$$\underset{\mathbf{x}' \in \mathbf{X}^-}{\text{argmin}} \ \min\{f(\mathbf{x}'), 0\} + C(\mathbf{x}', \mathbf{x}), \quad \text{subject to } C(\mathbf{x}', \mathbf{x}) \leq \delta_{\max}. \tag{5}$$

Given an original malicious app, an effective adversarial attack generally modifies a small portion of features with the low evasion cost. Let Eq. (5) returns an optimal solution $\mathbf{x}^*$ with $\text{sign}(f(\mathbf{x}^*)) = -1$, and a suboptimal solution $\tilde{\mathbf{x}}$ with $\text{sign}(f(\tilde{\mathbf{x}})) = -1$, we characterize the relationship between $\mathbf{x}^*$ and $\tilde{\mathbf{x}}$, and have

$$\min\{f(\mathbf{x}^*), 0\} = \min\{f(\tilde{\mathbf{x}}), 0\} = -1.$$

The difference between $\mathbf{x}^*$ and $\tilde{\mathbf{x}}$ can be simplified as the comparison between their evasion costs, i.e., argmin $C(\mathbf{x}', \mathbf{x})$. According to the definition of the evasion cost in Eq. (3), $C(\mathbf{x}', \mathbf{x}) \geq 0$. $C(\mathbf{x}', \mathbf{x}) = 0$ iff $\mathbf{x}' = \mathbf{x}$. $C(\cdot)$ is then strictly convex in $\mathbf{x}'$ and has a unique solution for this optimization problem. Therefore, $C(\mathbf{x}^*, \mathbf{x}) < C(\tilde{\mathbf{x}}, \mathbf{x})$, since $\mathbf{x}^*$ is an optimal attack. The evasion cost varies resting on the different levels of knowledge the attackers have about the targeted learning system. We formalize this relationship between $\mathbf{x}^*$ and $\tilde{\mathbf{x}}$ in the following lemma.

Lemma 3.1. *Suppose $\mathbf{x}^*$ is the optimal adversarial attack to Eq. (5), while $\tilde{\mathbf{x}}$ is suboptimal attack, s.t., $f(\mathbf{x}^*) < 0$ and $f(\tilde{\mathbf{x}}) < 0$. Then $C(\mathbf{x}^*, \mathbf{x}) < C(\tilde{\mathbf{x}}, \mathbf{x})$. That is, the optimal adversarial attack can access to the learning-based system with minimum evasion cost.*

According to this lemma, to perform an optimal adversarial attack, attacker may want to select the features that are easy to be manipulated (e.g., addition is generally easier than elimination) and to choose the features that have higher contributions to the classification problem. This inspires us to design a secure defense to combat the adversarial attack strategy.

4 DEFENSE FOR ADVERSARIAL ATTACKS

To be resilient against the adversarial attacks, in this paper, we aim to enhance the security of a learning-based system by enforcing attackers to manipulate a large number of features to evade the detection. Recall that, as discussed in Section 3, it may not be always feasible for attackers to generate a variant of malicious app by modifying a lot of features (i.e., with high evasion cost) while preserving its original semantics and functionalities. In this section, we present our two-step novel defense strategy as follows.

4.1 *SecCLS:* Novel Feature Selection Method to Construct More Secure Classifier

In adversarial settings, the importance of a feature from an attacker's perspective is: (i) its contribution to the classification problem, which is corresponding to the weight w trained by the classifier based on the training data, and (ii) its complexity being manipulated, that is, the manipulation cost c decided by feature type (e.g., permission vs. API call) and manipulation method (e.g., addition vs. elimination). Given ith-feature ($1 \leq i \leq d$) extracted from the dataset D, its importance can be defined as follow:

$$\mathcal{I}(i) \propto \frac{|w_i|}{c_i}, \tag{6}$$

wi——权重
ci——修改成本

the larger ... and the lower ..., the more important..

which implies that the larger the weight of the feature trained by the classifier and the lower the cost of the feature being manipulated, the more important the feature to the attackers. Clearly, the importance of a feature represents the possibility that an attacker may manipulate it in an adversarial attack.

The rationale to construct a more secure classifier against the adversarial attacks is to reduce the possibility of those important features being selected for model construction. In other words, those features the attackers tend to manipulate (i.e., features with higher values of $\mathcal{I}(i)$) may not present together in the learning model, which will intuitively force attackers to manipulate a larger number of other less important features (i.e., features with lower values of $\mathcal{I}(i)$) to evade the detection. In this way, the probability of each feature being selected for constructing a classification model is inversely proportional to its importance, that is, the more important the feature is to attackers, the less possible it will be selected to train the classifier. We formalize $\mathcal{P}(i)$, the probability of ith-feature being selected, as:

$$\mathcal{P}(i) \propto \frac{\lambda}{\mathcal{I}(i)}, \tag{7}$$

where λ is an adjustable parameter which can be empirically decided based on the training data. When substituting Eq. (6) into Eq. (7), the length of the probability is actually arbitrary long (e.g., $|w_i| = 0$). To normalize $\mathcal{P}(i)$, we further define $\mathcal{P}(i)$ as:

$$\mathcal{P}(i) = \lambda c_i (1 - \rho |w_i|), \tag{8}$$

where ρ ($0 < \rho < 1$) is a rescaling parameter to keep $\mathcal{P}(i)$ in the range of $(0, 1]$.

For the weight w_i of ith-feature, it can be calculated by the learning-based classifier in Eq. (2) trained on the dataset D. Provided that Eq. (2) is an optimization problem, based on the derivation and substitution, the weight vector for all features can be calculated as:

$$\mathbf{w} = \beta \mathbf{X} \xi, \tag{9}$$

$$s.t.\ \xi = (\beta \mathbf{X}^T \mathbf{X} + \gamma \mathbf{I})^{-1} \mathbf{f}, \tag{10}$$

where $\mathbf{f}$ can be solved through Eq. (2) using conjugate gradient descent method. We then further normalize each weight $|w_i|$ using *min-max normalization* [18] to the range of $[0, 1]$.

For the manipulation cost c_i of ith-feature, as discussed in Section 3, it can be estimated with respect to its feature type and the manipulation method. Considering that (1) feature addition is usually easier than elimination, and (2) compared with permissions and filtered intents in the manifest file, API calls and new-instances in dex file are relatively easier to be manipulated, Figure 2 illustrates different costs empirically decided for manipulating different kinds of features in our application.

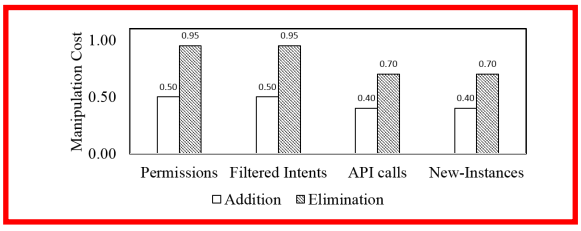


Figure 2: The manipulation costs determined by different feature types and manipulation methods.

这三点挺重要：特征空间；训练集；学习算法

Attacker may know completely, partially, or do not have any information of the targeted learning system about: (i) the feature space, (ii) the training data set, and (iii) the learning algorithm [33]. We would like to overestimate attackers' capabilities rather than underestimate them. Since this worst case provides a potential upper bound on the performance degradation suffered by the learning system under the adversarial attacks, it can be used as reference to evaluate the effectiveness of the learning system under the other limited attack scenarios. To conduct a well-crafted attack, we assume that attackers are capable to access the targeted learning system and may have perfect knowledge regarding the system. Therefore, they can use the methods such as *information gain* [18] or *max-relevance* [28] to calculate the information of each feature for the classification of malicious and benign apps respectively. Then they will be able to utilize those features that significantly contribute to benign apps classification for additions, and apply those ones that significantly contribute to malicious apps classification for eliminations.

As the above presentation, we can see that $\mathcal{P}(i) \in (0, 1]$, where the minimum is attained when the feature is most informative for the classification task or easy to be manipulated, and the maximum is attained when the feature has least contribution to the classification problem or is too costly to be manipulated. We then form the probability set for selecting features to construct a more secure classifier as:

$$\mathcal{P} = \{\mathcal{P}(1), \mathcal{P}(2), ..., \mathcal{P}(d)\}. \tag{11}$$

Given a pseudo random function $\mathcal{R}(.) \in (0, 1)$, the original feature vector of a given app $\mathbf{x}_i$ will be represented by an updated binary feature vector $\bar{\mathbf{x}}_i$:

$$\bar{\mathbf{x}}_i =< \bar{x}_{i1}, \bar{x}_{i2}, \bar{x}_{i3}, ..., \bar{x}_{id} >,$$

where

$$\bar{x}_{ij} = \begin{cases} x_{ij} & \mathcal{R}(.) \leq \mathcal{P}(j) \\ 0 & \text{otherwise} \end{cases}. \tag{12}$$

The proposed feature selection method for classifier construction is named *SecCLS*, whose implementation is given in Algorithm 1. Note that when $\mathcal{P}(i)$ $(1 \leq i \leq d)$ is with the same value for each feature, i.e., feature importances are evenly distributed, our proposed feature selection method *SecCLS* is approximate to random selection. Thus we can say, random feature selection method [5, 19] is a lower bound of *SecCLS*. Our proposed feature selection method *SecCLS* reduces the possibility of those features that attackers tend to manipulate, which will accordingly force attackers to manipulate a larger number of other features and thus be more resilient against their attacks. For computational complexity of *SecCLS*, to get weight vector $\mathbf{w}$ from Eq. (2) requires $O(d^3)$ queries, while to form cost vector $\mathbf{c}$ and calculate $\mathcal{P}$ both need $O(d)$. Since we formalize n apps as $\mathbf{X}$, each column of which is the d-dimensional feature vector, to get an updated training set $\bar{\mathbf{X}}$ from $\mathbf{X}$ requires $O(nd)$ updates.

By using *SecCLS*, after feature selection, the learning-based classifier in Eq. (2) can be updated as:

$$\underset{\bar{\mathbf{f}}, \mathbf{w}, \mathbf{b}; \xi}{\operatorname{argmin}} \frac{1}{2}||\mathbf{y} - \bar{\mathbf{f}}||^2 + \frac{1}{2\beta}\mathbf{w}^T\mathbf{w} + \frac{1}{2\gamma}\mathbf{b}^T\mathbf{b} + \xi^T(\bar{\mathbf{f}} - \bar{\mathbf{X}}^T\mathbf{w} - \mathbf{b}), \tag{13}$$

Algorithm 1: ***SecCLS*** – A novel feature selection method to construct more secure classifier.

Input: Training data set $D = \{\mathbf{x}_i, y_i\}_{i=1}^n$.
Output: $\bar{\mathbf{X}}$: updated training set based on the selected features.

```
Get weight vector w by the learning-based classifier in Eq. (2)
  trained on D;
Get manipulation cost vector c;
Calculate P = {P(1), P(2), ..., P(d)} using Eq. (8);
k = 1;
for k ≤ d do
    Get a pseudo random number from R(.);
    if R(.) ≤ P(k) then
        X̄_{k.} = X_{k.}
    else
        X̄_{k.} = 0
    end
    k++;
end
return X̄;
```

subject to $\bar{\mathbf{f}} = \text{sign}(f(\bar{\mathbf{X}}))$, where $\bar{\mathbf{f}}$ is the predicted label vector based on a feature set $\bar{\mathbf{X}}$. To solve the problem in Eq. (13), let

$$\mathcal{L}(\bar{\mathbf{f}}, \mathbf{w}, \mathbf{b}; \xi) = \frac{1}{2}||\mathbf{y} - \bar{\mathbf{f}}||^2 + \frac{1}{2\beta}\mathbf{w}^T\mathbf{w} + \frac{1}{2\gamma}\mathbf{b}^T\mathbf{b} + \xi^T(\bar{\mathbf{f}} - \bar{\mathbf{X}}^T\mathbf{w} - \mathbf{b}). \tag{14}$$

Based on the substitution and derivation from $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0$, $\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = 0$, $\frac{\partial \mathcal{L}}{\partial \xi} = 0$, $\frac{\partial \mathcal{L}}{\partial \bar{\mathbf{f}}} = 0$, we can get the more secure classifier as:

$$[\mathbf{I} + (\beta\bar{\mathbf{X}}^T\bar{\mathbf{X}} + \gamma\mathbf{I})^{-1}]\bar{\mathbf{f}} = \mathbf{y}. \tag{15}$$

4.2 *SecENS:* An Ensemble Learning Approach to Improve the Detection Accuracy

In the previous section, a novel feature selection method *SecCLS* is presented for constructing a more secure classifier against the adversarial attacks. To improve the system security while not compromising the detection accuracy, in this section, we further propose an ensemble learning approach called *SecENS* to aggregate a set of classifiers built using *SecCLS* to generate the final output for the detection.

Ensemble methods are a popular way to overcome instability and increase performance in many machine learning tasks [43], such as classification, clustering and ranking. An *ensemble* of classifiers is a set of classifiers whose individual decisions are combined in some way (e.g., by weighted or unweighted voting) to classify new samples, which is shown to be much more accurate than the individual classifiers that make them up [41]. Typically, an ensemble can be decomposed into two cascaded components: the first component is to create base classifiers with necessary accuracy and diversity; the second one is to aggregate all of the outputs of base classifiers into a numeric value as the final output of the ensemble. In general, base classifiers are generated by subsampling training set or input features (as done in boosting or bagging), manipulating the output targets, or injecting randomness in the learning algorithm [12].

In this paper, with certain accuracy of each individual classifier, we aim to diversify the classifiers that form the ensemble while also consider the integration of whole feature space. More specifically, the set of classifiers in the ensemble should follow two criteria: (1) the feature set used for building each classifier should differentiate from each other (i.e., *feature differentiation*), and (2) the ensemble should cover as many features as possible to assure the integration of whole feature space (i.e., *feature integration*). Therefore, we first construct each individual classifier using the proposed feature selection method *SecCLS* described in Section 4.1; then we follow the above two criteria and propose *SecENS* to aggregate a set of the constructed classifiers to generate the final output for the detection. We present *SecENS* beginning with the definitions of the above two criteria.

Feature Differentiation (denoted as f_D). Given two feature sets $\mathbf{F}^a \in \mathbb{R}^d$ and $\mathbf{F}^b \in \mathbb{R}^d$, which are selected using the proposed method *SecCLS* respectively, the differentiation between them can be defined as:

$$f_D(\mathbf{F}^a, \mathbf{F}^b) = 1 - J(\mathbf{F}^a, \mathbf{F}^b) = \frac{|\mathbf{F}^a \cup \mathbf{F}^b| - |\mathbf{F}^a \cap \mathbf{F}^b|}{|\mathbf{F}^a \cup \mathbf{F}^b|}. \tag{16}$$

Feature Integration (denoted as f_I). The feature integration of an ensemble is the percentage of features that are included in at least one of the base classifiers, which can be defined as follow:

$$f_I(\mathbf{F}) = \frac{|\bigcup_{k=1}^{K} \mathbf{F}^k|}{d}, \tag{17}$$

where the feature set $\mathbf{F} \in \mathbb{R}^d$ in the ensemble is aggregated by the feature sets $\mathbf{F}^k$ ($k = 1, 2, ..., K$) from base classifiers.

In *SecENS*, we employ boosting [12] during the training phase. Boosting works by sequentially applying a base classifier to train the updated weighted samples and aggregating all the outputs generated from the individual classifiers into the final prediction. At each iteration, the misclassified samples are assigned higher weights, so that at the next iteration, the classifier will focus more on learning those samples [16]. With the use of boosting, our proposed ensemble learning approach *SecENS* builds the ensemble by integrating both feature differentiation (f_D) and feature integration (f_I) to diversify the classifiers while preserve a significant integration of whole feature space. Algorithm 2 illustrates the implementation of the proposed *SecENS* in detail.

Algorithm 2: ***SecENS*** – An ensemble learning approach to improve the detection accuracy.

Input: Training data set $D = \{\mathbf{x}_i, y_i\}_{i=1}^n$; $\mathcal{W}$: weights of training apps; ε: error rate for each classifier; ζ: importance of each classifier; η_d: specified threshold of f_D; η_f: specified threshold of f_I; η_a: specified threshold of training accuracy.

Output: $\mathbf{f}$: the labels for the apps.

Initialize: $\mathcal{W}_1(i) = \frac{1}{n}$ for $i = 1, 2, ..., n$; $t = 0$;
Get training set $\bar{\mathbf{X}}^1$ and selected feature set $\mathbf{F}^1$ on D using *SecCLS*;
while *1* **do**
  $t++$;
  Train a base classifier using $\bar{\mathbf{X}}^t$;
  Get weak hypothesis $f_t: \bar{\mathbf{X}}^t \rightarrow \{-1, 1\}$;
  Calculate error rate of f_t:
  $\varepsilon_t \leftarrow \sum_{i=1}^n \mathcal{W}_t(i)[y_i \neq f_t(\bar{\mathbf{x}}_i^t)]$;
  Set $\zeta_t = \frac{1}{2}\ln(\frac{1-\varepsilon_t}{\varepsilon_t})$;
  Update $\mathcal{W}_{t+1}(i) = \frac{\mathcal{W}_t(i)\exp(-\zeta_t \bar{y}_i f_t(\bar{\mathbf{x}}_i))}{\sum_{i=1}^n \mathcal{W}_t(i)\exp(-\zeta_t \bar{y}_i f_t(\bar{\mathbf{x}}_i))}$ for $i = 1, 2, ..., n$;
  Calculate $f_I(\mathbf{F})$;
  Calculate the training accuracy (acc) of the ensemble based on $\mathbf{f} = \text{sign}(\sum_{i=1}^t \zeta_i f_i(\mathbf{X}))$;
  if $f_I(\mathbf{F}) \geq \eta_f$ and $acc \geq \eta_a$ **then**
    break;
  end
  Get $\bar{\mathbf{X}}^{t+1}$ and $\mathbf{F}^{t+1}$ on D using *SecCLS*;
  Calculate $f_D(\mathbf{F}^{t+1}, \mathbf{F}^j)$ for $j = 1, 2, ..., t$;
  while $\min\{f_D(\mathbf{F}^{t+1}, \mathbf{F}^j) \mid j = 1, 2, ..., t\} < \eta_d$ **do**
    Get $\bar{\mathbf{X}}^{t+1}$ and $\mathbf{F}^{t+1}$ on D using *SecCLS*;
    Calculate $f_D(\mathbf{F}^{t+1}, \mathbf{F}^j)$ for $j = 1, 2, ..., t$;
  end
end
return $\mathbf{f} = \text{sign}(\sum_{i=1}^t \zeta_i f_i(\mathbf{X}))$;

5 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, to empirically validate our developed system *SecureDroid* which integrates the proposed methods presented in Section 4, we present four sets of experimental studies using real sample collections obtained from Comodo Cloud Security Center: (1) In the first set of experiments, we evaluate the effectiveness of *SecureDroid* against different kinds of adversarial attacks; (2) In the second set of experiments, we assess the security of our proposed feature selection method *SecCLS* applied in the system *SecureDroid*; (3) In the third set of experiments, we further compare *SecureDroid* with other alternative defense methods; (4) In the last set of experiments, we evaluate the scalability of *SecureDroid* based on a larger sample collection.

5.1 Experimental Setup

5.1.1 Data Collection. The real sample collections we obtained from Comodo Cloud Security Center contain two sets: (1) The first sample set includes 8,046 apps (4,729 are benign apps, while the remaining 3,317 apps are malware including the families of Geinimi, GinMaster, DriodKungfu, Hongtoutou, FakePlayer, etc.). The extracted features from this sample set is with 926 dimensions, which include 104 permissions, 204 filtered intents, 330 API calls, and 288 new-instances. (2) The second dataset has larger sample collection containing 72,891 Android apps (40,448 benign apps and 32,443 malicious apps).

5.1.2 Evaluation Measures. To quantitatively validate the effectiveness of different methods in Android malware detection, we use the performance indices shown in Table 2.

5.1.3 Implementation of Different Adversarial Attacks. To thoroughly assess the security and detection accuracy of our developed

Table 2: Performance indices of Android malware detection

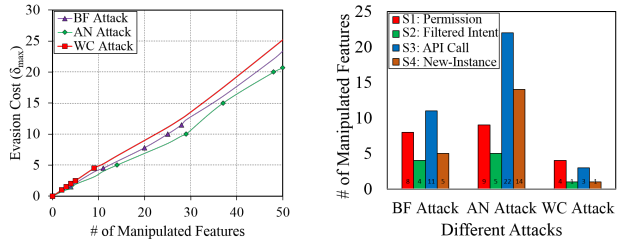
Indices	Description
TP	Number of apps correctly classified as malicious
TN	Number of apps correctly classified as benign
FP	Number of apps mistakenly classified as malicious
FN	Number of apps mistakenly classified as benign
Precision	$TP/(TP+FP)$
Recall/TPR	$TP/(TP+FN)$
ACC	$(TP+TN)/(TP+TN+FP+FN)$
F1	$2 \times Precision \times Recall/(Precision+Recall)$

system *SecureDroid* against a wide class of attacks, we define and implement three kinds of representative adversarial attacks [25, 26, 46] considering different skills and capabilities of attackers, which are presented as followings.

Brute-force (BF) attack. To implement such kind of attack, for each malicious app (i.e., $\mathbf{x}^+$) we would like to manipulate, we first use Jaccard similarity [18] to find its most similar benign app (i.e., $\mathbf{x}^-$) from the sample set. Given these two apps, the procedure begins with $\mathbf{x}^+$ and modifies features one at a time to match those of $\mathbf{x}^-$, until the malicious app is classified as benign or the evasion cost reaches to $\delta_{\max}$.

Anonymous (AN) attack. To simulate anonymous attack in which the defenders may have zero knowledge of what the attack is, we randomly manipulate some features for addition and some for elimination with the evasion cost of $\delta_{\max}$.

Well-crafted (WC) attack. In this adversarial setting, we use the wrapper-based approach [25, 46] to iteratively select a feature and greedily update this feature to incrementally increase the classification errors of the targeted learning system. Specifically, we first rank the features using methods such as *information gain* [18] to calculate their contributions to the classification problem. Then we conduct bi-directional feature selection, i.e., forward feature addition and backward feature elimination, to manipulate the malicious apps. At each iteration, using the attack model formulated in Eq. 5 (in Section 3.3) which encodes two competing objectives (i.e., maximizing the classification error while minimizing the evasion cost for optimal attacks), a feature will be either added or eliminated.



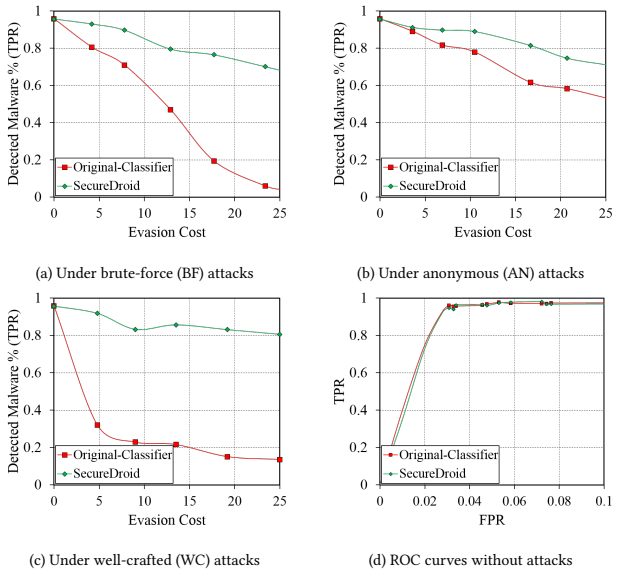
(a) Manipulated features No. vs. $\delta_{\max}$ (b) Manipulated features No. (under 50% *TPR*)

Figure 3: Effectiveness evaluation of different attacks.

To estimate the effectiveness of different attacks, we implement the above three kinds of attacks to access the *Original-Classifier* described in Section 2.3 and make its *TPR*s drop from 90% to 50%. For each attack, Figure 3.(a) shows the relations between the numbers of manipulated features and the corresponding evasion costs which also consider the complexity of different feature manipulations. Among these attacks, the well-crafted (WC) attack is the most effective strategy, since the evasion cost of this attack (also the number of features manipulated by this attack) is minimum when compromising the learning classifier into the same level, which can be seen in Figure 3.(b).

5.2 Evaluation of *SecureDroid* against Different Kinds of Adversarial Attacks

In this set of experiments, based on the first sample set described in Section 5.1, we validate the effectiveness of *SecureDroid* against above mentioned adversarial attacks. To estimate the reasonable evasion cost for attackers to perform the adversarial attacks, based on the first sample set, we explore the average number of features that each app possesses, which is 98. In general, 50% of the average number of features is considered as an extreme for the adversary to perform the attack. Based on these observations, we implement the above three kinds of attacks to access both *SecureDroid* and *Original-Classifier* with the manipulated features varying in {10%, 20%, 30%, 40%, 50%} of the average number of features (i.e., 98), whose corresponding evasion costs under different kinds of attacks are shown in Figure 4.(a)–(c) (X-axis). We randomly select 90% of the samples for training, while the remaining 10% is used for testing. We use these attacks to taint the malicious apps in the testing set respectively, and then assess the security of *SecureDroid* under different attacks with different evasion costs by comparison with the *Original-Classifier*. To implement *SecureDroid*, empirically we found that the parameters of $\lambda = 0.7$ and $\rho = 0.8$ in Eq. 8 are the best, and apply them to our problem throughout the experiments. To validate the detection performance of *SecureDroid* without attacks, we also perform 10-fold cross validations for evaluation. The experimental results are shown in Figure 4.



(a) Under brute-force (BF) attacks (b) Under anonymous (AN) attacks

(c) Under well-crafted (WC) attacks (d) ROC curves without attacks

Figure 4: Security evaluations of *SecureDroid* and *Original-Classifier* under brute-force (BF) attacks, anonymous (AN) attacks, well-crafted (WC) attacks, and without attacks.

Under attacks. From Figure 4.(a)–(c), we can see that *SecureDroid* can significantly enhance security compared to the *Original-Classifier*, as its performance decreases more elegantly against increasing evasion costs, especially in the scenarios of BF attack and WC attack. In the BF attack, the *TPR* of *Original-Classifier* drops to 5.99% with evasion cost $\delta_{\max}$ of 23.4 (i.e., modifying 50 features), while *SecureDroid* retains the *TPR* at 70.06% with the same evasion cost. In the WC attack, the performance of *Original-Classifier* is compromised to a great extent with *TPR* of 13.62% under the evasion cost of 25.2; instead, *SecureDroid* can significantly bring the detection system back up to the desired performance level: the *TPR*s of *SecureDroid* are actually never lower than 80.00% even with increasing evasion costs. This demonstrates that *SecureDroid* which integrates our proposed methods is resilient against the most effective attack strategy (i.e., WC attack) among the three representative adversarial attacks. In the AN attack, which is simulated under defenders have zero knowledge of what the attack is and by randomly injecting or removing features from the malicious apps, *SecureDroid* also outperforms the *Original-Classifier*, which can retain the average *TPR* at 85.16% with different evasion costs.

Without attacks. Figure 4.(d) shows the ROC curves of the 10-fold cross validations for *Original-Classifier* and *SecureDroid* without any attacks. From Figure 4.(d), we can see that, though *SecureDroid* is designed to be resilient against different kinds of adversarial attacks, its detection performance is as good as the *Original-Classifier* in the absence of attacks.

The experimental results and above analysis demonstrate that *SecureDroid* can effectively enhance security of the learning-based classifier without compromising the detection accuracy, even attackers may have different knowledge about the targeted learning system. Based on these properties, *SecureDroid* can be a resilient solution in Andriod malware detection.

5.3 Evaluation of *SecCLS* applied in *SecureDroid*

In this section, based on the same training and testing datasets in Section 5.2, we further validate the effectiveness and significance of our proposed feature selection method *SecCLS* in building a more secure classifier. We compare *SecureDroid* which applies *SecCLS* to select features for each base classifier with ensemble of random feature selection (denoted as *ERFS*) that uses random feature selection method to construct base classifiers [5, 19], in the settings of under attacks and without attacks. As illustrated in Section 5.1.3, since well-crafted (WC) attack is the most effective attack strategy among those three, we evaluate the *SecureDroid* and *ERFS* under such kind of attacks. The experimental results are shown in Table 3.

From Table 3, we can observe that, (1) **Under attacks:** *ERFS* can somehow be resilient against the attack (with *TPR* of 85.63%) when the evasion cost is small ($\delta_{\max} = 4.8$, modifying 10 features). However, with the increasing evasion costs, the detection performance of *ERFS* drops drastically (e.g., its *TPR* drops to 16.47% when the evasion cost $\delta_{\max}$ is 25.2 corresponding to manipulating 50 features). In contrast, *SecureDroid* using *SecCLS* for feature selection can significantly enhance security, as its performance decreases more elegantly against increasing evasion costs and its *TPR*s are actually never lower than 80.00% with different evasion costs. The reason behind this is that *SecCLS* integrated in *SecureDroid* reduces

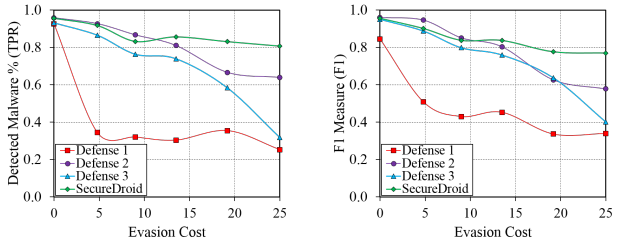
Table 3: Comparison of *SecureDroid* with *SecCLS* and *ERFS* with random feature selection against well-crafted attacks (UnderAtt) and without attacks (NonAtt).

		NonAtt	UnderAtt [$\delta_{\max}$ (features modified)]				
			4.8(10)	9.0(20)	13.5(30)	19.2(40)	25.2(50)
ERFS	TPR	0.9072	0.8563	0.5045	0.4326	0.2934	0.1647
	ACC	0.9230	0.9354	0.7888	0.7559	0.6981	0.6509
	F1	0.9072	0.9167	0.6647	0.5953	0.4465	0.2813
SecureDroid	TPR	0.9566	0.9177	0.8323	0.8563	0.8308	0.8069
	ACC	0.9634	0.9168	0.8665	0.8621	0.8019	0.8106
	F1	0.9559	0.9015	0.8380	0.8375	0.7768	0.7795

the possibility of selecting those features attackers tend to manipulate, i.e., to achieve same attack utility, *SecCLS* will force attackers to modify larger number of features compared with random feature selection method. (2) **Without attacks:** *SecureDroid* also performs better than *ERFS* in the absence of attacks (i.e., about 4-5% higher detection accuracy). This is because, compared with *ERFS* which randomly assigns equal probability for each feature being selected, *SecureDroid* applying *SecCLS* method is capable to retain majority of the features for each individual classifier and thus assure its detection accuracy in the absence of attacks.

5.4 Comparisons of *SecureDroid* with Other Alternative Defense Methods

In this set of experiments, we further examine the effectiveness of *SecureDroid* against the adversarial attacks (i.e., well-crafted attack as it shows most effective) by comparisons with other popular defense methods, including (1) ***feature evenness*** (denoted as ***Defense1***) which enables the *Original-Classifier* to learn more evenly-distributed feature weights using the method proposed in [23]; (2) ***classifier retraining*** (denoted as ***Defense2***) which follows Stackelberg game theories [7, 8, 17, 34] and models the attack as a vector $\boldsymbol{\theta}$ to modify the training data set $\mathbf{X}$ where the *Original-Classifier* is retrained [34, 38]; (3) ***classifier built on reduced feature set*** (denoted as ***Defense3***) which carefully selects a subset of features based on the generalization capability of the *Original-Classifier* and its security against data manipulation applying the method proposed in [46]. The experimental results are reported in Figure 5.



(a) *TPR*s under attacks (b) F1 measures under attacks

Figure 5: Comparisons of different defense methods.

From Figure 5, we can see that *SecureDroid* significantly outperforms the other defense models (i.e., *Defense1–3*) against the

well-crafted attacks. Although ***Defense2*** (i.e., classifier retraining) performs slightly better than *SecureDroid* when the evasion costs $\delta_{\max} \in \{4.8, 9.0\}$ (i.e., modifying 10 and 20 features), the difference is not statistically significant. In fact, the retrained model modifies the training data distribution approximate to the testing space through the attack model $\boldsymbol{\theta}$. After modifying a large number of features in the malicious apps, the model tends to produce a distribution that is very close to that of the benign apps. In this case, the retrained model may not be able to differentiate benign and malicious apps accurately. From Figure 5, we also observe that as the evasion cost $\delta_{\max}$ increases, the performance of the retrained model suffers a great drop-off. For *Defense1* and *Defense3*, their performances (*TPR*s and *F1* measures in Figure 5) sharply degrade when evasion cost increases. For ***Defense1***, the weight evenness merely exploits the information of the classifier's feature weights while ignoring manipulation costs of different features; for ***Defense3***, the model is built on a carefully selected feature subset, whose robustness could be compromised when attackers manipulate a certain number of these features.

5.5 Scalability Evaluation of *SecureDroid*

In this section, based on the second sample set with larger size described in Section 5.1 which consists of 72,891 apps (32,443 malicious and 40,448 benign), we systematically evaluate the performance of our developed system *SecureDroid*, including scalability and detection effectiveness. We first evaluate the training time of *SecureDroid* with different sizes of the training sample sets. Figure 6 presents the scalability of our developed system. We can observe that as the size of the training data set increases, the running time for our detection system is quadratic to the number of training samples. When dealing with more data, approximation or parallel algorithms could be developed. Figure 7 shows the detection stability of *SecureDroid* against the adversarial attacks (i.e., well-crafted attacks) and in the absence of attacks, with different sizes of sample sets. From the results, we can conclude that our developed system *SecureDroid* can enhance security of machine learning based detection, and is feasible in practical use for Android malware detection against adversarial attacks.

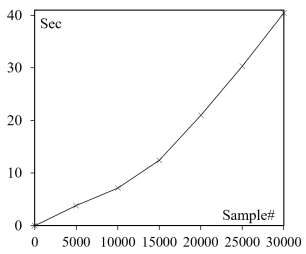


Figure 6: Scalability evaluation of *SecureDroid*.

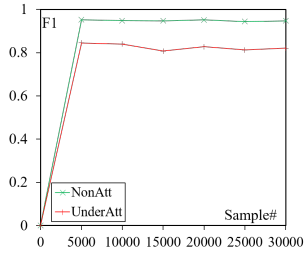


Figure 7: Stability evaluation of *SecureDroid*.

6 RELATED WORK

In recent years, there are ample researches that show machine learning-based systems can actively adjust their behaviors to combat the adversarial attacks in some cybersecurity domains. In these systems, the detection methods can be generally divided into four categories: Stackelberg game theories [7, 8, 17, 34], feature operations [14, 24, 46], retraining frameworks [15, 25, 38], and ensemble classifier systems [5, 11, 23]. To apply Stackelberg game theories, Bruckner et al. [7] first presented the interaction between the learner and the adversary as a static game, and explored the adversarial properties to find the equilibrate prediction model; they then further simulated the interaction as a Stackelberg competition, and derived an optimization problem to determine the solution of this game [8]. Wang et al. [34] modeled the adversary action as it controlling a vector $\boldsymbol{\alpha}$ to modify the training data set $\mathbf{X}$, and transformed the classifier into a convex optimization problem. More recently, feature operation methods have also been proposed to counter some kinds of adversarial data manipulations, such as feature deletion [14], feature clustering [24], feature reduction [46], etc. In addition, retraining frameworks are becoming more and more widely applied to boost the resilience of learning algorithms through: (1) adding adversarial samples to the training data that evade the previously computed classifier [15, 25], and (2) manipulating the training data distribution that its distribution is matched to the test data [38]. Though these theories and approaches are promising, most of them make strong assumptions about the structure of the data (e.g., adversarial samples) or the attack model that are likely impractical for Android malware detection problems. To improve the security of machine learning under generic settings, some research efforts have been devoted to multiple classifier systems. Kolcz et al. [23] applied averaging method resting on random subsets of reweighted features to produce a linear ensemble classifier. Biggio et al. [5] built a multiple-classifier system to improve the robustness of the classifier through bagging, and the random subspace method. Debarr et al. [11] explored randomization to generalize learning model by randomly choosing dataset or features, and estimated parameters that fit the data best. In these ensemble learning systems, randomization is the main method for feature selection.

Different from the existing works, in this paper, we present a novel feature selection method for constructing more secure classifier by taking advantage of Android malware detection domain-related knowledge, and then an ensemble learning approach is further introduced to combine the classifiers built using the proposed feature selection method to retain the detection accuracy. The proposed method is independent from the skills and capabilities of the attackers and can be readily applied in other malware detection tasks.

7 CONCLUSION

In this paper, we explore the security of machine learning in Android malware detection to understand how feature selection impacts on the security of a learning-based classifier. Our study considers different importances of the features associated with their contributions to the classification problem and manipulation costs to the adversarial attacks. We propose a novel feature selection method *SecCLS* which reduces the possibility to select those features attackers tend to manipulate and thus helps to construct more secure classifier. To improve the system security while not compromising the detection accuracy, we further propose an ensemble

learning approach *SecENS* by aggregating the individual classifiers that are constructed using the proposed *SecCLS*. Accordingly, we develop a system called ***SecureDroid*** which integrates both *SecCLS* and *SecENS* to enhance security of machine learning-based Android malware detection. Comprehensive experiments on the real sample collections from Comodo Cloud Security Center are conducted to validate the effectiveness of *SecureDroid*. The results demonstrate that our feature selection method *SecCLS* is more resilient to disrupt the feature manipulations, and *SecureDroid* can improve the security against the adversarial attacks even that attackers are with different skills and capabilities or have different knowledge about the targeted learning system. Our proposed secure-learning paradigm can also be readily applied to other malware detection tasks.

ACKNOWLEDGMENTS

The authors would also like to thank the anti-malware experts of Comodo Security Lab for the data collection, as well as the helpful discussions and supports. This work is supported by the U.S. National Science Foundation under grant CNS-1618629 and WVU Senate Grants for Research and Scholarship (R-16-043).

REFERENCES

[1] Mohammed S. Alam and Son T. Vuong. 2013. Random Forest Classification for Detecting Android Malware. In *GreenCom-iThings-CPSCom*. 663–669.
[2] Android. 2017. Application Fundamentals. In *https://developer.android.com/guide/components/fundamentals.html*.
[3] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. 2010. The security of machine learning. *Machine Learning* 81, 2 (2010), 121–148.
[4] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. 2006. Can machine learning be secure?. In *ASIACCS '06*. 16–25.
[5] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2010. Multiple Classifier Systems for Robust Classifier Design in Adversarial Environments. *International Journal of Machine Learning and Cybernetics* 1, 1 (2010), 27–41.
[6] Battista Biggio, Giorgio Fumera, and Fabio Roli. 2014. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering* 26, 4 (2014), 984–996.
[7] Michael Bruckner, Christian Kanzow, and Tobias Scheffer. 2012. Static prediction games for adversarial learning problems. *Journal of Machine Learning Research* 13, 1 (2012), 2617–2654.
[8] Michael Bruckner and Tobias Scheffer. 2011. Stackelberg games for adversarial prediction problems. In *KDD '11*. 547–555.
[9] Lingwei Chen, William Hardy, Yanfang Ye, and Tao Li. 2015. Analyzing File-to-File Relation Network in Malware Detection. In *WISE '15 International Conference on Web Information Systems Engineering*. 415–430.
[10] Lingwei Chen and Yanfang Ye. 2017. SecMD: Make Machine Learning More Secure Against Adversarial Malware Attacks. In *AI '17 Australasian Joint Conference on Artificial Intelligence*. 76–89.
[11] Dave Debarr, Hao Sun, and Harry Wechsler. 2013. Adversarial Spam Detection Using the Randomized Hough Transform-Support Vector Machine. In *ICMLA '13*. 299–304.
[12] Thomas G. Dietterich. 2000. Ensemble methods in machine learning. *Multiple Classifier Systems* 1 (2000), 1–15.
[13] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A survey of mobile malware in the wild. In *SPSM '11*. 3–14.
[14] Amir Globerson and Sam Roweis. 2006. Nightmare at Test Time: Robust Learning by Feature Deletion. In *ICML '06*. 353–360.
[15] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *ICLR '15*.
[16] Haixiang Guo, Yijing Li, Yanan Li, Xiao Liu, and Jinling Li. 2016. BPSO-Adaboost-KNN ensemble learning algorithm for multi-class imbalanced data classification. *Engineering Applications of Artificial Intelligence* 49 (2016), 176–193.
[17] Nika Haghtalab, Fei Fang, Thanh H. Nguyen, Arunesh Sinha, Ariel D. Procaccia, and Milind Tambe. 2016. Three Strategies to Success: Learning Adversary Models in Security Games. In *IJCAI'16*. 308–314.
[18] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques 3rd Edition*. Morgan Kaufmann, Waltham, MA, USA.
[19] Tin Kam Ho. 1998. The Random Subspace Method for Constructing Decision Forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 8 (1998), 832–844.
[20] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. 2016. Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs. In *WIW '16*.
[21] Shifu Hou, Aaron Saas, Yanfang Ye, and Lifei Chen. 2016. DroidDelver: An Android Malware Detection System Using Deep Belief Network Based on API Call Blocks. In *WAIM '16*. 54–66.
[22] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In *KDD '17*. 1507–1515.
[23] Aleksander Kolcz and Choon Hui Teo. 2009. Feature Weighting for Improved Classifier Robustness. In *CEAS '09 Sixth conference on email and anti-spam*.
[24] Bo Li and Yevgeniy Vorobeychik. 2014. Feature cross-substitution in adversarial classification. In *NIPS'14*. 2087–2095.
[25] Bo Li, Yevgeniy Vorobeychik, and Xinyun Chen. 2016. A General Retraining Framework for Scalable Adversarial Classification. In *NIPS 2016 Workshop on Adversarial Training*.
[26] Daniel Lowd and Christopher Meek. 2005. Adversarial Learning. In *KDD '05*. 641–647.
[27] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. In *IEEE Symposium on Security and Privacy (SP)*. 582–597.
[28] Hanchuan Peng, Fuhui Long, and C. Ding. 2005. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE TPAMI* 27, 8 (2005), 1226–1238.
[29] Don Reisinger. 2013. Android, iOS combine for 91 percent of market. In *https://www.cnet.com/news/android-ios-combine-for-91-percent-of-market/*.
[30] Fabio Roli, Battista Biggio, and Giorgio Fumera. 2013. Pattern recognition systems under attack. In *CIARP '13*. 1–8.
[31] Anthony Scarsella and William Stofega. 2017. Worldwide Smartphone Forecast, 2017–2021. In *http://www.idc.com/getdoc.jsp?containerId=US42366217*.
[32] Michael Spreitzenbarth. 2016. Current Android Malware. In *https://forensics.spreitzenbarth.de/android-malware/*.
[33] Nedim Šrndic and Pavel Laskov. 2014. Practical Evasion of a Learning-Based Classifier: A Case Study. In *SP '14*. 197–211.
[34] Fei Wang, Wei Liu, and Sanjay Chawla. 2014. On Sparse Feature Attacks in Adversarial Learning. In *ICDM '14*. 1013–1018.
[35] Paul Wood. 2015. Internet Security Threat Report 2015. In *Symantec, California*.
[36] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *ASIAJCIS '12 Proceedings of the 2012 Seventh Asia Joint Conference on Information Security*.
[37] Wen-Chieh Wu and Shih-Hao Hung. 2014. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In *RACS '14 Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*.
[38] Yifan Wu, Tianshu Ren, and Lidan Mu. 2016. Importance Reweighting Using Adversarial-Collaborative Training. In *NIPS 2016 Workshop*.
[39] Jianlin Xu, Yifan Yu, Zhen Chen, Bin Cao, Wenyu Dong, Yu Guo, and Junwei Cao. 2013. MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Sci. Technol.* 18, 4 (2013), 418–427.
[40] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. *ESORICS European Symposium on Research in Computer Security* 8712 (2014), 163–182.
[41] Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. 2009. SBMDS: an interpretable string based malware detection system using SVM ensemble with bagging. *Journal in Computer Virology* 5 (2009), 283–293.
[42] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *ACM Computing Surveys (CSUR)* 50, 3 (2017), 41.
[43] Yanfang Ye, Tao Li, Qingshan Jiang, Zhixue Han, and Li Wan. 2009. Intelligent File Scoring System for Malware Detection from the Gray List. In *KDD '09*. 1385–1394.
[44] Yanfang Ye, Tao Li, Shenghuo Zhu, Weiwei Zhuang, Egemen Tas, Umesh Gupta, and Melih Abdulhayoglu. 2011. Combining File Content and File Relations for Cloud Based Malware Detection. In *KDD '11 Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 222–230.
[45] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-Sec: deep learning in android malware detection. In *SIGCOMM '14 Proceedings of the 2014 ACM conference on SIGCOMM*. 371–372.
[46] Fei Zhang, Patrick P. K. Chan, Battista Biggio, Daniel S. Yeung, and Fabio Roli. 2015. Adversarial Feature Selection Against Evasion Attacks. *IEEE Transactions on Cybernetics* 46, 3 (2015), 766–777.
[47] Min Zhao, Fangbin Ge, Tao Zhang, and Zhijian Yuan. 2011. AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android. In *ICICA '11 International Conference on Information Computing and Applications*. 158–166.