2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

DroidEye: Fortifying Security of Learning-based Classifier against Adversarial Android Malware Attacks

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

Shouhuai Xu
Department of Computer Science
University of Texas at San Antonio
shouhuai.xu@utsa.edu

***Abstract*—To combat the evolving Android malware attacks, systems using machine learning techniques have been successfully deployed for 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 first extract a set of features from the Android applications (apps) and represent them as binary feature vectors; with these inputs, we then explore the security of a generic learning-based classifier for Android malware detection in the presence of adversaries. To harden the evasion, we first present *count featurization* to transform the binary feature space into continuous probabilities encoding the distribution in each class (either benign or malicious). To improve the system security while not compromising the detection accuracy, we further introduce *softmax function with adversarial parameter* to find the best trade-off between security and accuracy for the classifier. Accordingly, we develop a system named *DroidEye* which integrates our proposed method for Android malware detection. Comprehensive experiments on the real sample collection from Comodo Cloud Security Center are conducted to validate the effectiveness of *DroidEye* against adversarial Android malware attacks. Our proposed secure-learning paradigm is also applicable for other detection tasks, such as spammer detection in social media.**

I. Introduction

Smart phones have been widely used in people's everyday life to perform tasks such as social networking, online shopping, financial management, and entertainment. Android, designed as an open, free and programmable operation system for smart phone platforms, has dominated the current market share [22]. However, due to its large market share and open source ecosystem of development, Android not only attracts the developers for producing legitimate apps, but also attackers to disseminate malware (short for ***ma***licious soft***ware***) onto legitimate users to disrupt the mobile operations. Today, a lot of Android malware (e.g., GhostPush, Obad, RevMob, and BankBot) is released on the markets; it's reported [17] that over $750,000$ new malicious Android apps were discovered in the first quarter of 2017. 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. As a result, the detection of Android malware is of major concern to both the anti-malware industry and researchers.

IEEE/ACM ASONAM 2018, August 28-31, 2018, Barcelona, Spain
978-1-5386-6051-5/18/$31.00 © 2018 IEEE

To combat the evolving Android malware attacks, systems applying machine learning techniques have been developed for automatic Android malware detection in recent years [10]–[12], [24], [27], [28]. In these systems, based on different feature representations (e.g., Application Programming Interface (API) calls [11]), dynamic behaviors [24]), various kinds of classification approaches, such as Support Vector Machine (SVM) [8], and Deep Neural Network [10], [11], are used for model construction to detect malicious apps. The effectiveness of the learning-based systems relies on the assumption that training data and testing data follow the same underlying distribution. However, this hypothesis is likely to be violated by an adversary who may carefully manipulate the input data (e.g., injecting API calls in a dead code, hiding a feature from the app while not affecting the intrusive functionality) to make the classifier produce maximum false negative (i.e., maximumly misclassifying malware as benign) [1].

In this paper, we investigate the adversarial Android malware attacks and aim to fortify security of learning-based classifier against such attacks. As conducting an adversarial attack that needs to manipulate a lot of features while not compromising the malicious functionalities may not always be feasible, in the adversarial point of view, to perform a practical attack, attackers intend to minimize the manipulations (i.e., modify the features as less as possible) to bypass the detection. 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 [5], [7]. In this work, we first extract a set of features (i.e., permissions, filtered intents, application attributes, API calls, new-instances, and exceptions) from the Android apps and represent them as binary feature vectors; with these inputs, we then explore the security of a learning-based classifier for Android malware detection in the presence of adversaries. To harden the evasion, we first present *count featurization* [14], [23] to transform the binary feature space into continuous probabilities. To improve the system security while not compromising the detection accuracy, we further introduce *softmax function* [3] *with adversarial parameter* for model construction. Accordingly, we develop a system called ***DroidEye*** which integrates the proposed method to fortify security of learning-based classifier against adversarial Android malware attacks. The system architecture of *DroidEye* is shown in Figure 1, which has the following major traits:

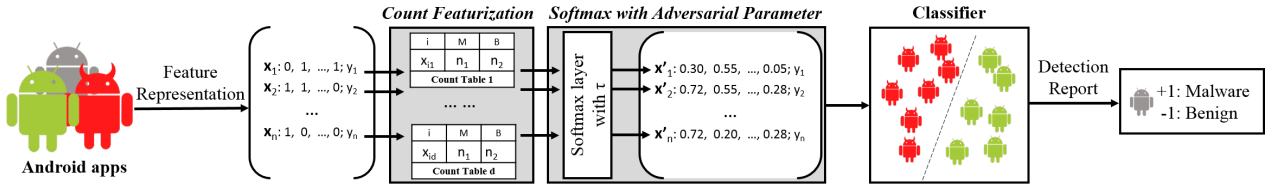


Figure 1: An overview of system architecture of *DroidEye*. In the system, the collected apps are first represented as d-dimensional binary feature vectors. To harden the evasion, *count featurization* is used to transform each binary feature vector $\mathbf{x}_i$ to a continuous feature vector $\mathbf{x}'_i$; then *softmax function with adversarial parameter* is introduced to find the best trade-off between security and accuracy for the classifier. For a new app, after feature representation, it will be predicted as either benign or malicious using the classifier.

- *Count featurization for feature transformation to harden the evasion:* To evade the detection, an adversary may manipulate the features to the original malicious app (i.e., either add or eliminate binaries in the vector) to make the classifier produce false negative (i.e., misclassify malware as benign). To be resilient against such attacks, we present count featurization to transform the binary feature space into continuous probabilities encoding the distribution in each class (either benign or malicious) to reduce the adversarial gradient of the learning model.
- *Softmax function with adversarial parameter for model construction to fortify system security while not compromising detection accuracy:* Based on the count-featurized probability vectors, we introduce softmax function (i.e., normalized exponential function) with adversarial parameter to find the best trade-off between security and accuracy for the classifier, by tuning the adversarial parameter.
- *A practical and resilient system for Android malware detection in the presence of adversaries:* Based on a real sample collection from Comodo Cloud Security Center (including $6,745$ malicious apps and $8,059$ benign apps), we develop a system *DroidEye* which integrates our proposed method to fortify the security of learning-based classifier against adversarial Android malware attacks. A series of comprehensive experiments are conducted and the results demonstrate that *DroidEye* can bring the detection system back up to the desired performance level against different kinds of adversarial attacks, including L_0 attacks, L_∞ attacks and anonymous attacks.

The rest of the paper is organized as follows. Section II defines the problem of learning-based classifier for Android malware detection. Section III discusses the adversarial attacks. Section IV introduces our proposed method in detail. Section V systematically evaluates the effectiveness of *DroidEye*. Section VI discusses the related work. Finally, Section VII concludes.

II. 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 provide a brief introduction to learning-based classifier resting on the feature representations of Android apps with preliminaries.

A. Preliminaries

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 retains information about its structure [10]: *Activities*, *Services*, *Broadcast Receivers*, and *Content Providers*. The components are first configured using a set of *application attributes* to set default values for corresponding elements (e.g., whether allow the app to reset user data). The actions of each component are further specified through *filtered intents* which declare the types of intents it can respond to (e.g., through filtered intents, an activity can initiate a phone call). The manifest file also contains a list of *permissions* requested by the app to perform functions (e.g., access Internet). Since permissions, filtered intents, and application attributes 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 developed with Java can be converted into Dalvik executable (dex) files, run on the Dalvik Virtual Machine (DalvikVM)[1]. The dex file contains compiled code written for Android and can be interpreted as user-implemented methods and classes by the DalvikVM, in which *API calls* are used to access operating system functionality and resources, and *new-instances* are used to create new instances of classes from operating system classes. The dex file also utilizes *exceptions* to indicate conditions that an app may want to catch. Therefore, API calls, new-instances, and exceptions 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 features.

In this paper, we perform static analysis on the collected Android apps and extract the above features to represent

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

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 fortify the security properties of learning-based classifier 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.

B. 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 II-A, these six sets of features (***S1–S6*** shown in Table I) include: permissions (*S1*), filtered intents (*S2*), and application attributes (*S3*) from manifest files, API calls (*S4*), new-instances (*S5*), and exceptions (*S6*) from dex files.

Table I: Illustration of extracted features

	Features	Examples
Manifest	*S1:* Permissions	*INTERNET*
	S2: Filtered Intents	*action.MAIN*
	S3: Application Attributes	*debuggable*
Dex	*S4:* API calls	*containsHeader*
	S5: New-Instances	*util/HashMap*
	S6: Exceptions	*SecurityException*

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 features in *S1–S6* in dataset D. Each app can then be represented by a binary feature vector:

$$\mathbf{x}_i = \begin{pmatrix} 0 \\ \dots \\ 1 \\ \dots \\ 0 \\ \dots \\ 1 \\ \dots \\ 0 \\ \dots \\ 0 \\ \dots \end{pmatrix} \rightarrow \begin{matrix} \text{READ\_PHONE\_STATE} \\ \dots \\ \text{vending.INSALL\_REFERER} \\ \dots \\ \text{allowClearUserData} \\ \dots \\ \text{getSimSerialNumber} \\ \dots \\ \text{Landroid/app/ProgressDialog} \\ \dots \\ \text{ArithmeticException} \\ \dots \end{matrix} \begin{matrix} \} \textit{S1} \\ \\ \} \textit{S2} \\ \\ \} \textit{S3} \\ \\ \} \textit{S4} \\ \\ \} \textit{S5} \\ \\ \} \textit{S6} \\ \end{matrix}$$

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, $x_{ij} = 0$).

C. Learning-based Classifier for Android Malware Detection

The problem of learning-based classifier for Android malware detection can be stated in the form of: $f : \mathcal{X} \rightarrow \mathcal{Y}$ 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:

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

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 learning-based classifier can be formalized as an optimization problem:

$$\underset{\mathbf{f},\mathbf{w},\mathbf{b}}{\text{argmin}}\ \mathcal{L}(\mathbf{y}, \mathbf{f}) + \underbrace{\beta||\mathbf{w}|| + \gamma||\mathbf{b}||}_{\mathcal{R}(f)}, \tag{2}$$

subject to Eq. (1), where $\mathbf{y}$ is the labeled information vector, $\mathcal{L}(\mathbf{y}, \mathbf{f})$ is a loss function, $\mathcal{R}(f)$ is a regularization term to avoid overfitting, β and γ are the regularization parameters. Note that Eq. (2) is a typical learning-based classifier. Without loss of generality, the equation can be transformed into different learning models depending on the choices of loss function and regularization terms [8], such as SVM.

III. Adversarial Attacks

In Android malware detection, a learning-based classifier is to detect malicious apps based on the trained classification model and prevent them from interfering users' smart phones. In 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*) [1]. In this paper, we focus on the integrity attack, also called adversarial attack. To conduct an adversarial attack, attackers would manipulate the features of a malicious app to evade the detection. As described in Section II-B, given an Android app, after feature extraction, it can be represented by a binary feature vector. Then a typical manipulation can be either adding a binary (i.e., set 0 to 1) or eliminating a binary (i.e., set 1 to 0) in the vector. Compared with feature addition, 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. To perform an adversarial attack, attackers need to take consideration of the evasion cost. In this respect, the 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., $\hat{\mathbf{x}} \in \mathcal{X}^-$) with the minimal evasion cost, which can be formulated as:

$$\underset{\hat{\mathbf{x}} \in \mathcal{X}^-}{\text{argmin}}\ \min\{f(\hat{\mathbf{x}}), 0\} + \mathcal{C}(\hat{\mathbf{x}}, \mathbf{x}), \tag{3}$$

where $\mathcal{C}(\hat{\mathbf{x}}, \mathbf{x})$ is the evasion cost (e.g., the number of binaries that are changed from $\mathbf{x}$ to $\hat{\mathbf{x}}$ by attackers). For binary feature space, since L_0 attacks [6], [19] and L_∞ attacks [9], [16], [30] are the most popular among all the adversarial attacks towards the learning-based classifiers [4], we here introduce

the representative L_0 attack model [6] and L_∞ attack model [9] as follows.

A. L_0 Attack Model

L_0 attacks [4] measure $\mathcal{C}(\hat{\mathbf{x}}, \mathbf{x})$ using L_0 distance (i.e., the number of features that are altered in an malicious app). As an L_0 attack model, *AdvAttack* [6] uses the wrapper-based approach [30] to iteratively select a feature and greedily update this feature to incrementally increase the classification errors of the targeted learning system. Specifically, it ranks the features using *Max-Relevance* algorithm [29] to calculate their contributions to the classification problem. Then it conducts bi-directional feature selection, i.e., forward feature addition and backward feature elimination, to manipulate the malicious samples. At each iteration, using the attack model formulated in Eq. (3), a feature will be either added or eliminated. In this paper, we implement AdvAttack to conduct L_0 attack for our further investigation.

B. L_∞ Attack Model

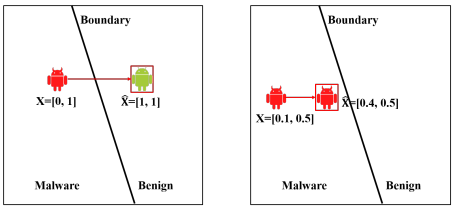
L_∞ attacks [4] measure $\mathcal{C}(\hat{\mathbf{x}}, \mathbf{x})$ using L_∞ distance (i.e., the maximum change to any of the features in an malicious app). As an L_∞ attack model, given a malicious app $\mathbf{x}$, Fast Gradient Sign Method (FGSM) [9] sets

$$\hat{\mathbf{x}} = \mathbf{x} + \varepsilon \cdot \text{sign}(\nabla_{\mathbf{x}} \mathcal{L}(f(\mathbf{x}), y)), \tag{4}$$

where $\mathcal{L}$ is the loss function used in classifier training, y is the target label for $\mathbf{x}$, and ε is a constant parameter. Intuitively, for each feature x_i, FGSM uses the gradient of the loss function to determine in which direction the feature's value should be increased or decreased to minimize the loss function. To apply FGSM to the binary feature space in our application, we further define a threshold θ to adjust $\hat{\mathbf{x}}$ so that $\hat{x}_i = \{0, 1\}$, i.e., if $\hat{x}_i \geq \theta$, then $\hat{x}_i = 1$; otherwise $\hat{x}_i = 0$. In this paper, we implement FGSM to conduct L_∞ attack for our further investigation.

IV. Defense against Adversarial Attacks

By performing AdvAttack described in Section III-A, attackers may autonomously add a feature in the app (i.e., set 0 to 1 in the vector). For example, they can add permissions in the manifest file without influence on other existing functionalities; they can also inject API calls in the methods which will be never called by any invoke instructions in the dex file. Figure 2(a) shows an example that attackers can successfully generate a variant ($\hat{\mathbf{x}} = [1, 1]$) to evade the detection by injecting a feature in the original malicious app (denoted as $\mathbf{x} = [0, 1]$). But from the defenders' point of view, if the binary feature space is featurized into continuous space of each feature value being $0 \leq x \leq 1$, the actual gradient of the feature addition or elimination available to the attackers may be significantly squashed. If adversarial gradients are low, crafting adversarial attacks becomes more difficult because small feature manipulations will not induce high output variations for the learning model [21], which thus makes the model more resilient against the adversarial attacks. As shown in Figure 2(b), with the same manipulation from $\mathbf{x}$ to $\hat{\mathbf{x}}$, the step towards the boundary is sufficiently shortened in the continuous feature space, which makes the evasion fail. This intuition of *gradient masking* [20] inspires us to design a secure defense with count featurization [14] to combat the adversarial attacks.



(a) Binary feature space (b) Continuous feature space

Figure 2: Defenses in different feature spaces.

A. Proposed Defense

Count featurization is originally motivated by the objective of reducing training time on data that contains categorical features by feeding learning algorithms with a limited subset of the collected data combined with historical collections from much larger amounts of data [14], [23]. The general idea of this technique is to featurize the data with the conditional probability of the class given the frequency (i.e., the number of times) a feature value was observed with each class, instead of directly using the value of a categorical feature [14]. Given a binary feature vector of an app $\mathbf{x}$, to perform count featurization, count tables for each feature are first aggregated on the original dataset. The conditional probabilities are then calculated directly from the count tables as defined below.

Definition 4.1: **Count table** is designed per feature in $\mathbf{x}$. It maintains the number of malicious apps for each feature value (denoted as $M(x_i)$) and the number of benign apps for each feature value (denoted as $B(x_i)$); it therefore encodes each feature's propensity to malware and benign apps.

Definition 4.2: To count-featurize a binary feature vector $\mathbf{x} = \langle x_1, x_2, ..., x_d \rangle$, **count featurization** projects each of its features with the conditional probabilities calculated from the count tables, i.e., $\mathbf{x}' = \langle \mathcal{P}(M(x_1)|x_1), ..., \mathcal{P}(M(x_d)|x_d) \rangle$, where $\mathcal{P}(M(x_i)|x_i) = M(x_i)/(M(x_i)+B(x_i))$ from the row matching x_i in the corresponding count table.

This is a simplified version of the count featurization function, which is particularly valuable when the features are of high cardinality [14]. Considering that each feature only has two values (i.e., 1 and 0) in our application, this potentially is at the cost of reducing predictive accuracy. To preserve each feature's informative property, we formulate a softmax function [3] to convert conditional probabilities into more effective action probabilities for model construction. The softmax function for feature x_i is given by

$$\bar{\mathcal{P}}(x_i) = \frac{\exp(\mathcal{P}(M(x_i)|x_i)/\tau)}{\sum_{k \in \{M(x_i), B(x_i)\}} \exp(\mathcal{P}(k|x_i)/\tau)}, \tag{5}$$

where τ is an adjustment parameter that plays a critical role to actively keep the trade-off between security and accuracy

for the classifier trained on the count-featurized probability vectors. In adversarial settings, we refer to this adjustment parameter as the *adversarial parameter*. The higher the adversarial parameter of softmax function is, the more ambiguous and secure its action probabilities will be (i.e., when $\tau \to +\infty$, all the probabilities are close to 0.5), whereas the smaller τ is, the more discrete and informative its probabilities will be (i.e., when $\tau \to 0^+$, the probabilities are close to 1 or 0) [21]. Therefore, based on the softmax function with adversarial parameter in Eq. (5), the final probability vector for $\mathbf{x}$ can be formulated as

$$\mathbf{x}' = \langle \bar{\mathcal{P}}(x_1), \bar{\mathcal{P}}(x_2), ..., \bar{\mathcal{P}}(x_d) \rangle. \tag{6}$$

Figure 3 shows an example of an app $\mathbf{x}$ and its count-featurized vector $\mathbf{x}'$ with $\tau = 0.5$.

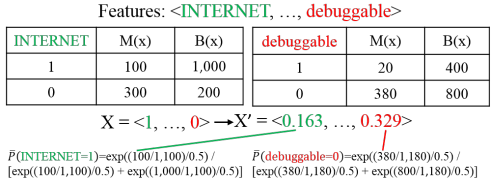


Figure 3: An example of count featurization.

An adversary-aware learning system for Android malware detection should (1) relatively consistently predict the correct labels for the manipulated apps, as well as (2) significantly display good accuracy on benign apps [21], [26]. Different from the previous work towards this goal which substantially performed model regularization [6], data retraining [9], [16], [25], or feature reduction [15], [30], we adapt count featurization to improve the security of the learning model while leaving model, training data, and feature sets unchanged. It's recalled that the benefit of count featurization in our application is intuitive as the probabilities ranging in $[0, 1]$ encode additional distribution information about each class, in addition to simply providing an app's feature existences, permitting more secure and accurate learning. To implement the defense, called *DroidEye*, we add a count featurization layer and a softmax layer with adversarial parameter τ in front of the learning model shown in Figure 1, which count-featurizes the binary feature vector $\mathbf{x}$ for each app into continuous vector $\mathbf{x}'$. The learning model predicts the class for a given app by training on count-featurized conditional probabilities. Note that, when classifying a new app, the adversarial parameter τ should be configured as a low value (e.g., $\tau = 1$) to make the predictions more accurate.

Algorithm 1 illustrates the implementation of the proposed defense (denoted as *DroidEye*) in detail. Since *DroidEye* has not changed the original model and training data, the only impact on computational complexity is limited for count featurization, requiring $O(nd)$ queries, which ensures that the learning model can still take advantage of large dataset to achieve the good performance.

Algorithm 1: *DroidEye* – Classifier with count featurization against adversarial Android malware attacks.

Inputs: Training data set $D = \{\mathbf{x}_i, y_i\}_{i=1}^n$; τ: adversarial parameter
Outputs: $\mathbf{f}$: the labels for the apps
Formulate count tables for features: $M(\mathbf{X})$ and $B(\mathbf{X})$
$i = 1$
for $i \leq n$ **do**
  Calculate $\langle \mathcal{P}(M(x_1)|x_1), ..., \mathcal{P}(M(x_d)|x_d) \rangle$
  Calculate $\mathbf{x}'_i = \langle \bar{\mathcal{P}}(x_1), \bar{\mathcal{P}}(x_2), ..., \bar{\mathcal{P}}(x_d) \rangle$
  $\mathbf{x}_i = \mathbf{x}'_i$
end for
Use conjugate gradient descent method to solve: $\underset{\mathbf{X},\mathbf{w},\mathbf{b}}{\operatorname{argmin}}\ \mathcal{L}(\mathbf{y}, f(\mathbf{X})) + \beta||\mathbf{w}|| + \gamma||\mathbf{b}||$
return $\mathbf{f} = \operatorname{sign}(f(\mathbf{X}))$

B. Theoretical Analysis

The adversary generally takes two steps to craft the adversarial attack: (1) evaluate the sensitivity of class change to each input feature, and (2) use the sensitivity information to select a set of manipulations among the input features [21]. In the attacks (e.g., FGSM) discussed in Section III, the sensitivity of the model to the feature manipulations is primarily evaluated through adversarial gradient, which is defined as the gradient difference between the adversarial attack and the original malware:

$$\nabla G = \nabla \mathcal{L}(f(\hat{\mathbf{x}}), y) - \nabla \mathcal{L}(f(\mathbf{x}), y). \tag{7}$$

The higher adversarial gradient denotes that crafting adversarial attacks is relatively easier as small feature manipulations will induce high output variation of the learning model [21]. The adversarial gradient will not vanish unless $\nabla_{\mathbf{x}} \mathcal{L}(f(\mathbf{x}), y)$ becomes zero, which is impractical [18]. But count featurization can significantly reduce ∇G to the small feature manipulations.

Again, our defense using continuous probability vectors by count featurization benefits from the additional knowledge found in the apps. The additional knowledge encodes the relative distributions of malware and benign apps, which prevents the models from fitting too tightly to the feature existences, and contributes to a more stable while still accurate feature representations around training data. On the contrary, the adversary may only manage to add or eliminate a small number of features to craft the attacks $\hat{\mathbf{x}}$, which may have limited impact on the actual probability distributions and data structure, that is, based on the same feature manipulations, roughly

$$\nabla \mathcal{L}(f(\hat{\mathbf{x}}'), y) - \nabla \mathcal{L}(f(\mathbf{x}'), y) < \nabla \mathcal{L}(f(\hat{\mathbf{x}}), y) - \nabla \mathcal{L}(f(\mathbf{x}), y). \tag{8}$$

Actually when the probabilities $\mathbf{x}'$ are all smoothed to be close to 0.5, $\nabla G_{\mathbf{x}'}$ would be significantly approaching 0. If the small feature manipulations cannot induce the evasion, the adversary may have to manipulate a larger number of features to achieve

the goal. Considering the evasion cost, and the app's original functionalities, this may not be always feasible.

Note that the count featurization is controlled by an adversarial parameter τ in softmax, which is capable of further adjusting the trade-off between the smoothness and accuracy of the learning model. Here, we further quantify the continuous feature space's smoothness to the input $\mathbf{x}$ by its Jacobian Matrix [21]. We use $\bar{\mathcal{P}}_i(x)$ to denote the probability of feature x to be with class i ($i \in \{malware, benign\}$), and let $G(x) = \exp(M(x)/\tau) + \exp(B(x)/\tau)$. Its formulation of component (i, j) at adversarial parameter τ is:

$$\begin{aligned}\left.\frac{\partial \bar{\mathcal{P}}_i(x)}{\partial x_j}\right|_\tau &= \frac{\partial}{\partial x_j}\left(\frac{\exp(M(x)/\tau)}{\exp(M(x)/\tau)+\exp(B(x)/\tau)}\right) \\ &= \frac{1}{G^2(x)}\left(\frac{\partial \exp(M(x)/\tau)}{\partial x_j}G(x) - \exp(M(x)/\tau)\frac{\partial G(x)}{\partial x_j}\right) \\ &= \frac{\exp(M(x)/\tau)\exp(B(x)/\tau)}{\tau G^2(x)}\left(\frac{\partial(\exp(M(x)-\exp(B(x))}{\partial x_j}\right)\end{aligned} \tag{9}$$

Since $M(x)$ and $B(x)$ are fixed values for each feature, and the component values are inversely proportional to τ, the increasing τ will essentially reduce the values of all the components of Jacobian matrix. This analysis illustrates that count featurization resting on high settings of τ reduces the model sensitivity to small feature manipulations. When τ is well tuned, the model may also preserve the reasonable generalization ability. The empirical analysis will be given in Section V.

V. Experimental Results and Analysis

In this section, we present three sets of experimental studies to empirically validate our developed system *DroidEye*. The real sample collection we obtained from Comodo Cloud Security Center contains $14,804$ apps ($8,059$ are benign apps, while the remaining $6,745$ apps are malware including the families of Geinimi, GinMaster, DriodKungfu, etc.) with 812 features, including 105 permissions, 68 filtered intents, 8 application attributes, 330 API calls, 259 new-instances, and 42 exceptions. We randomly select 90% of the samples for training, while the remaining 10% is used for testing.

To thoroughly assess the security and detection accuracy of *DroidEye* against a wide class of attacks, we implement three kinds of representative adversarial attacks including AdvAttack (in Section III-A), FGSM (in Section III-B), and an anonymous attack (ANAttack) by randomly manipulating some features for addition and elimination to simulate the attack in which the defenders may have zero knowledge of what the attack is. To quantitatively validate the effectiveness of different methods in Android malware detection, we use the performance indices shown in Table II.

A. Evaluation of DroidEye with Different Adversarial Parameter Values

In this set of experiments, we evaluate how different settings of the adversarial parameter τ in the count featurization function may influence the performance of our developed system *DroidEye*. Note that the adversarial parameter is set to 1 when count-featurizing the testing apps. That is, τ only impacts on model training. It's recalled that the adversarial parameter is the key to adjust the trade-off between the smoothness and accuracy of the learning model. Therefore, the objective here is to identify the optimal training adversarial parameter for *DroidEye* resting on our data collection. Here, we specifically explore AdvAttack with different numbers of manipulated features to taint the malicious apps in the testing set, and repeat the experiments by measuring the adversarial parameter τ varying in $\{0.1, 0.5, 1, 1.5, 2, 5, 6, 7, 10\}$. The experimental results are shown in Figure 4.

Table II: Performance indices of malware detection

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

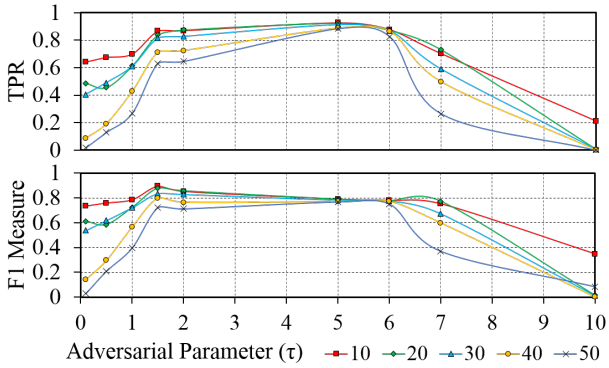
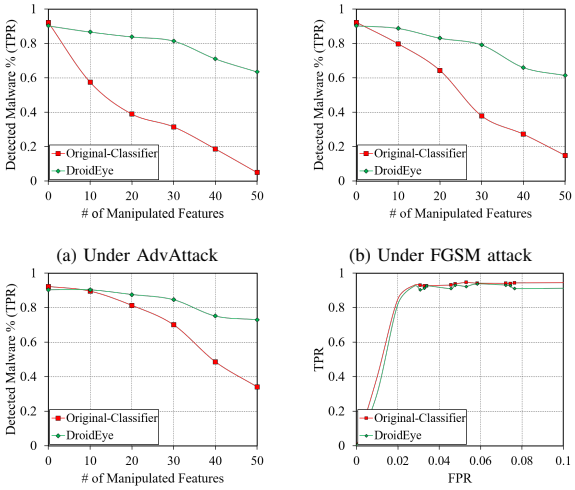


Figure 4: Evaluation of *DroidEye* with different τ under AdvAttack with number of manipulated features varying from 10 to 50.

From Figure 4, we can see that: (1) when $\tau \to 0^+$, the learning model is fairly vulnerable to the adversarial attacks, since the probability values in the feature space are extremely close to 1 or 0; increasing the parameter generally increases the *TPR*s while making adversarial evasion harder; (2) there is a turning point after the *TPR*s reach the highest (around $\tau = 5.5$); as $\tau \to 10$, the *TPR*s suffer from a drastic drop for all the probability values are approaching 0.5, which makes the features too ambiguous to discriminate malware from the benign apps. Observations validate our theoretical analysis in Section IV-A and Section IV-B. To fortify the security of the learning model while not compromising the detection accuracy, the optimal adversarial parameter should be linked to both *precision* and *recall*. In Figure 4, we can observe *F1* measures at $\tau = 1.5$ outperform the others with the highest average value (i.e., an average of 0.8254). Hence in the following experiments, we will formalize *DroidEye* based on the setting of $\tau = 1.5$.

B. Evaluation of DroidEye against Different Attacks

In this section, we validate the effectiveness of *DroidEye* against above mentioned adversarial attacks. We learn a linear SVM (denoted as *Original-Classifier*) as the learning-based classifier to facilitate our empirical analysis. To estimate the impact of feature manipulations on both *DroidEye* and *Original-Classifier*, we implement the above three kinds of attacks with the number of manipulated features varying in $\{10, 20, 30, 40, 50\}$, and then assess the security of *DroidEye* under different attacks by comparisons with *Original-Classifier*. To validate the detection accuracy of *DroidEye* without attacks, we also perform 10-fold cross validations for evaluation. The experimental results are shown in Figure 5.



(a) Under AdvAttack (b) Under FGSM attack

(c) Under ANAattack (d) ROC curves without attacks

Figure 5: Security evaluations of *DroidEye* and *Original-Classifier* under AdvAattack, FGSM attack, ANAattack, and without attacks.

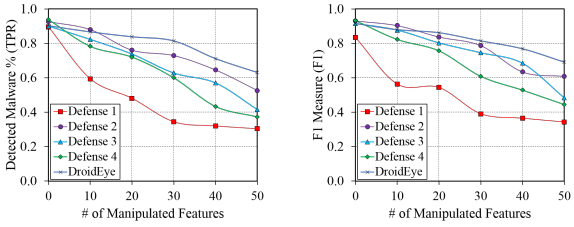
Security. Figure 5(a)–(c) signify that *DroidEye* can significantly enhance security compared to the *Original-Classifier*, as its performance decreases more elegantly against increasing manipulated features, especially in the scenarios of FGSM attack and AdvAttack. In the FGSM attack, the *TPR* of *Original-Classifier* drops to 14.94% with 50 manipulated features, while *DroidEye* retains the *TPR* at 61.37% with the same feature manipulations. In the AdvAttack, the performance of *Original-Classifier* is compromised to a greater extent with *TPR* of 4.98% with 50 features manipulated; instead, *DroidEye* can significantly bring the detection system back up to the desired performance level: the average *TPR* of *DroidEye* are actually stay around 77.00%. This demonstrates that *DroidEye* which devises our proposed count featurization is indeed resilient against those representative attack strategies. In the ANAttack, which is simulated under defenders have zero knowledge of what the attack is and by randomly injecting or removing features from the malicious apps, *DroidEye* also outperforms the *Original-Classifier*, which can retain the average *TPR* at 82.12% with different feature manipulations.

Accuracy. Figure 5(d) shows the ROC curves of the 10-fold cross validations for *Original-Classifier* and *DroidEye* without any attacks, from where we can see *DroidEye* is not only resilient against adversarial attacks, but its detection accuracy (an average 0.9210 *TPR* at 0.0525 *FPR*) is also as good as the *Original-Classifier* in the absence of attacks.

The experimental results and above analysis demonstrate that *DroidEye* can effectively fortify 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, *DroidEye* can be a resilient solution in Android malware detection.

C. Comparisons of DroidEye with Other Representative Defense Methods

In this set of experiments, we further examine the effectiveness of *DroidEye* against the adversarial attacks by comparisons with other popular defense methods, including (1) *feature evenness* (*Defense1*) which enables the *Original-Classifier* to learn more evenly-distributed feature weights through feature reweighting [13]; (2) *classifier retraining* (*Defense2*) which retrains the *Original-Classifier* using the adversarial examples [16], [25]; (3) *adversarial feature selection* (*Defense3*) which selects a subset of features based on the generalization capability of *Original-Classifier* and its security against data manipulation [30]; (4) *distillation* (*Defense4*) which applies the soft labels for training through softmax function devised in distillation layer [21]. As illustrated in Section V-B, AdvAttack is the most effective attack strategy among those three. Thus here we evaluate different defense methods under such kind of attacks. The experimental results are reported in Figure 6.



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

Figure 6: Comparisons of different defense methods.

From Figure 6, we can see that *DroidEye* significantly outperforms the other defense models (i.e., *Defense1–4*) against AdvAttack. Although *Defense2* performs slightly better than *DroidEye* with 10 features manipulated, the difference is not statistically significant. In fact, the retrained model modifies the training data distribution approximate to the testing space through the attack examples. 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 6, we also observe that as manipulated features increase, the performance of the retrained model suffers from a great drop-off. For *Defense1*, *Defense3*, and *Defense4*, their performances (*TPR*s and *F1* measures in Figure 6) sharply degrade when manipulated features increase. 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; for *Defense4* ($T = 1$), soft labels in training have limited impact on the linear learning classifier with only two outputs. From the results, we can conclude that *DroidEye* can fortify security of learning-based classifier, and is feasible in practical use for Android malware detection.

VI. Related Work

In adversarial settings, the detection methods can be generally divided into four categories: model modifications [6], [21], feature operations [15], [30], retraining frameworks [9], [16], [25], and ensemble classifier systems [2], [13]. The model modifications, either performing regularization [6] or adding new layers [21], suffer from limited success or expensive computation. More recently, feature operation methods have also been proposed to counter some kinds of adversarial data manipulations, such as feature clustering [15], feature reduction [30], etc. In addition, retraining frameworks are becoming more and more widely applied to boost the resilience of learning algorithms through adding adversarial samples [9], [16], or manipulating the training data distribution [25]. To improve the security of machine learning under generic settings, some research efforts have been devoted to multiple classifier systems [2], [13]. Different from the existing works, in this paper, we present a count featurization method to secure classifier by transforming the binary feature space into continuous, which leaves model and training data unchanged. The proposed defense is independent from the skills and capabilities of the attackers, and cost-effective to be realized.

VII. Conclusion

In this paper, we present count featurization for feature transformation to benefit from low gradient of feature manipulation available to an adversarial attack, and introduce softmax function with adversarial parameter to keep the trade-off between the security and accuracy of the model. Accordingly, we develop a system called *DroidEye* which integrates our proposed method to fortify security of learning-based Android malware detection. Comprehensive experiments on the real sample collections from Comodo Cloud Security Center are conducted to validate the effectiveness of *DroidEye*. The results demonstrate that *DroidEye* can improve the security against the adversarial attacks while not compromising the detection performance. Our proposed secure-learning paradigm can also be readily applied to other detection tasks, such as spammer detection in social media.

Acknowledgement

The authors would also like to thank the anti-malware experts of Comodo Security Lab for the data collection. This work is supported by the U.S. National Science Foundation under grants CNS-1618629 and CNS-1814825, WV Higher Education Policy Commission Grant (HEPC.dsr.18.5), and WVU Research and Scholarship Advancement Grant (R-844).

References

[1] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 2010.
[2] B. Biggio, G. Fumera, and F. Roli. Multiple Classifier Systems for Robust Classifier Design in Adversarial Environments. *IJMLC*, 2010.
[3] C. M. Bishop. Pattern Recognition and Machine Learning. 2006.
[4] N. Carlini and D. Wagner. Towards Evaluating the Robustness of Neural Networks. In *S&P*, 2017.
[5] L. Chen, S. Hou, and Y. Ye. SecureDroid: Enhancing Security of Machine Learning-based Detection against Adversarial Android Malware Attacks. In *ACSAC*, 2017.
[6] L. Chen and Y. Ye. SecMD: Make Machine Learning More Secure Against Adversarial Malware Attacks. In *AI*, 2017.
[7] L. Chen, Y. Ye, and T. Bourlai. Adversarial Machine Learning in Malware Detection: Arms Race between Evasion Attack and Defense. In *EISIC*, 2017.
[8] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. *TDSC*, 2017.
[9] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. In *ICLR*, 2015.
[10] S. Hou, A. Saas, L. Chen, and Y. Ye. Deep4MalDroid: A Deep Learning Framework for Android Malware Detection Based on Linux Kernel System Call Graphs. In *WIW*, 2016.
[11] S. Hou, A. Saas, Y. Ye, and L. Chen. DroidDelver: An Android Malware Detection System Using Deep Belief Network Based on API Call Blocks. In *WAIM*, 2016.
[12] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In *KDD*, 2017.
[13] A. Kolcz and C. H. Teo. Feature Weighting for Improved Classifier Robustness. In *CEAS*, 2009.
[14] M. Lecuyer, R. Spahn, R. Geambasu, T. Huang, and S. Sen. Pyramid: Enhancing Selectivity in Big Data Protection with Count Featurization. In *S&P*, 2017.
[15] B. Li and Y. Vorobeychik. Feature cross-substitution in adversarial classification. In *NIPS*, 2014.
[16] B. Li, Y. Vorobeychik, and X. Chen. A General Retraining Framework for Scalable Adversarial Classification. In *NIPS Workshop*, 2016.
[17] C. Lueg. In *https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day*, 2017.
[18] A. Nokland. Improving Back-Propagation by Adding an Adversarial Gradient. In *arXiv:1510.04189*, 2015.
[19] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z B. Celik, and A. Swami. The Limitations of Deep Learning in Adversarial Settings. In *EuroS&P*, 2016.
[20] N. Papernot, P. McDaniel, A. Sinha, and M. Wellman. Towards the Science of Security and Privacy in Machine Learning. In *arXiv*, 2016.
[21] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks. In *S&P*, 2016.
[22] D. Reisinger. In *https://www.cnet.com/news/android-ios-combine-for-91-percent-of-market/*, 2013.
[23] A. Srivastava, A. C. König, and M. Bilenko. Time Adaptive Sketches (Ada-Sketches) for Summarizing Data Streams. In *SIGMOD*, 2016.
[24] W. Wu and S. Hung. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. *RACS*, 2014.
[25] Y. Wu, T. Ren, and L. Mu. Importance Reweighting Using Adversarial-Collaborative Training. In *NIPS Workshop*, 2016.
[26] W. Xu, D. Evans, and Y. Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *arXiv:1704.01155*, 2017.
[27] Y. Ye, L. Chen, S. Hou, W. Hardy, and X. Li. DeepAM: a heterogeneous deep learning framework for intelligent malware detection. *KAIS*, 2018.
[28] Y. Ye, T. Li, D. Adjeroh, and S S. Iyengar. A Survey on Malware Detection Using Data Mining Techniques. *ACM CSUR*, 50(3), 2017.
[29] Y. Ye, T. Li, Q. Jiang, Z. Han, and L. Wan. Intelligent file scoring system for malware detection from the gray list. In *KDD*, 2009.
[30] F. Zhang, P. P. K. Chan, B. Biggio, D. S. Yeung, and F. Roli. Adversarial Feature Selection Against Evasion Attacks. *IEEE Transactions on Cybernetics*, 2015.