An Investigation on Fragility of Machine Learning Classifiers in Android Malware Detection

Husnain Rafiq, Nauman Aslam, Biju Issac and Rizwan Hamid Randhawa
Department of Computer and Information Sciences
Northumbria University
Newcastle, United Kingdom
{husnain.rafiq, nauman.aslam, biju.issac, rizwan.randhawa}@northumbria.ac.uk

***Abstract*—Machine learning (ML) classifiers have been increasingly used in Android malware detection and countermeasures for the past decade. However, ML-based solutions are vulnerable to adversarial evasion attacks. An attacker can craft a malicious sample carefully to fool an underlying pre-trained classifier. In this paper, we highlight the fragility of the ML classifiers against adversarial evasion attacks. We perform mimicry attacks based on Oracle and Generative Adversarial Network (GAN) against these classifiers using our proposed methodology. We use static analysis on Android applications to extract API-based features from a balanced excerpt of a well-known public dataset. The empirical results demonstrate that among ML classifiers, the detection capability of linear classifiers can be reduced as low as 0% by perturbing only up to 4 out of 315 extracted API features. As a countermeasure, we propose *TrickDroid*, a cumulative adversarial training scheme based on Oracle and GAN-based adversarial data to improve evasion detection. The experimental results of cumulative adversarial training achieves a remarkable detection accuracy of up to 99.46% against adversarial samples.**

I. Introduction

The arm-race between Android security companies and malware developers seems to be enduring. Machine learning has been demonstrated as the core element of Android malware detection by many researchers; however, it is vulnerable to evasion attacks [1], [2]. The adversarial evasion attacks are primarily dependent on the attacker's insight to defender's feature set of training data [3]. The detection model makes a prediction based on ranked features that can be a piece of sensitive information for the attacker. The attacker can make a slight change into any of the top-ranked features to generate an adversarial sample [4], [5]. However, such attacks are based on the domain knowledge of the attacker. There can be three possible knowledge levels for the attacker. The first and the best case is full knowledge (FK), where the attacker has access to the training data and knows about the underlying classifier. The second is limited knowledge (LK), where the attacker has no knowledge about the underlying classifier, however, still has access to the training data. The third case is no knowledge (NK), where the attacker neither has access to training data nor the knowledge about the underlying classifier. Although the Android malware detectors can hide the underlying model, however, there are many publicly available Android malware datasets that can help the attacker to get insights into the training data [6]. So there is a large gap to fill in research for adversarial evasion detection considering the publicly available datasets while designing a sophisticated Android malware detector.

Given the AMD, authors in [7] were the first to discuss the problem of evasion attacks. They performed multiple lightweight evasion attacks and were able to evade 50% of commercial malware analysis tools. Aydogan et al. [8] applied genetic programming to formulate evasion attacks and were able to evade 33% of Android malware on commercial antivirus tools. Meng et al. [9] proposed a technique to automatically generate Android malware samples to test multiple classifiers and antivirus tools and achieved an average evasion rate of 80%. Grosse et al. [5] crafted adversarial Android malware samples to evade deep neural networks and were able to evade almost 80% of the adversarial samples. Calleja et al. [10] proposed LagoDroid, a tool to generate evasion attacks against an existing Android malware detector called RevealDroid [11] and achieved an evasion rate of 98%. We propose a more lightweight and practical evasion attack methodology using feature injection achieving up to 100% evasion rate. We inject the top features of benign Android applications into malicious samples and test the ML-classifiers.

Most of the existing techniques to evade malware classifiers were either based on the gradient information or manual crafting of rules till 2017 [12]. However, later it was established that Generative Adversarial Networks (GANs) could also be used to automatically generate adversarial examples to trick ML classifiers. Zhang et al. [13] proposed AndrOpGAN, a technique to generate adversarial examples of Android malware that achieved an evasion rate of 99% against four malware detectors. Furthermore, Li et al. [14] proposed a technique based on bi-objective GANs to generate a novel adversarial examples attack method against Android malware classifiers. Salman et al. [15] used GANs to harden the security of Android malware detectors through intents based features. Taheri et al. [16] used five different evasion attack models on Android malware classifiers and used GANs to formulate a countermeasure against evasion attacks. The authors in [16] claim that GAN based methods improve the evasion detection of Android malware up to 50%. Millar et al. proposed DanDroid, a novel model to classify obfuscated and unobfuscated Android benign and malicious applications by using GANs. In our work, we feed GANs with malicious data to generate

synthetic data that mimic the real malicious applications. In the proposed model, we have used classifier-two sample test (C2ST) to evaluate the generator of GAN in addition to the expectancy loss and accuracy of both generator and discriminator.

We red-flag the fragility of ML classifiers such as support vector machine (SVM), logistic regression (LR), perceptron (PT), decision tree (DT), random forest (RF) and xgboost (XGB) to compare their effective candidacies for the AMD. We have performed Oracle and Generative Adversarial Network (GAN) based adversarial attacks against a practical dataset called *Drebin* that is publicly available [17]. We propose a technique to generate adversarial evasion examples that fool the classifiers mentioned above. It has been demonstrated that the linear classifiers SVM, LR, and PT are least impressive in contrast to their ensemble counterparts in the AMD for Android. Since there is no silver bullet defence against evasion attacks, therefore, only proactively knowing the attacker's manipulations could be cardinal to a robust defence strategy [3]. This is where adversarial training comes that has proved to be an effective proactive defence [18]. We propose a robust adversarial training scheme called ***TrickDroid*** based on cumulative adversarial training of ensemble classifiers on Oracle and GAN based adversarial data to improve evasion detection. Finally, we compare our results with adversarial training of individual Oracle and GAN based attacks and adversarial training.

The rest of this paper is organised as follows. Section II explains the proposed methodology for evasion attacks, the experimental results followed by analysis are presented in Section III and Section IV concludes this paper.

II. Proposed Attacks Methodology

Our proposed methodology of evasion attacks is illustrated in Figure 1 which shows the key components of the system. In the feature extraction module, we reverse engineer the Android applications to extract API-based features. The extracted features are further used to train multiple ML classifier models. To evade the trained classifiers, we generate code injection and GAN-based adversarial data in the adversarial samples generation module. The adversarial samples are further tested on the existing pre-trained classifiers. Finally, we perform adversarial training to harden the security of Android malware classifiers against adversarial evasion attacks.

A. Dataset and Feature Extractor

In this study, we use Drebin [17] as a benchmark dataset. The dataset is composed of 5560 malicious and 213,453 benign applications. We randomly select 5600 benign applications to balance the dataset. Furthermore, we reverse engineer the Android application packages (APKs) in the dataset to extract java source code. APKs are decompiled in the form of *.dex* and then transformed into *.jar* files. The *.jar* files are then disassembled into java source code in order to extract features. Static analysis is applied on the reverse-engineered code to extract API-based features from the Android applications.

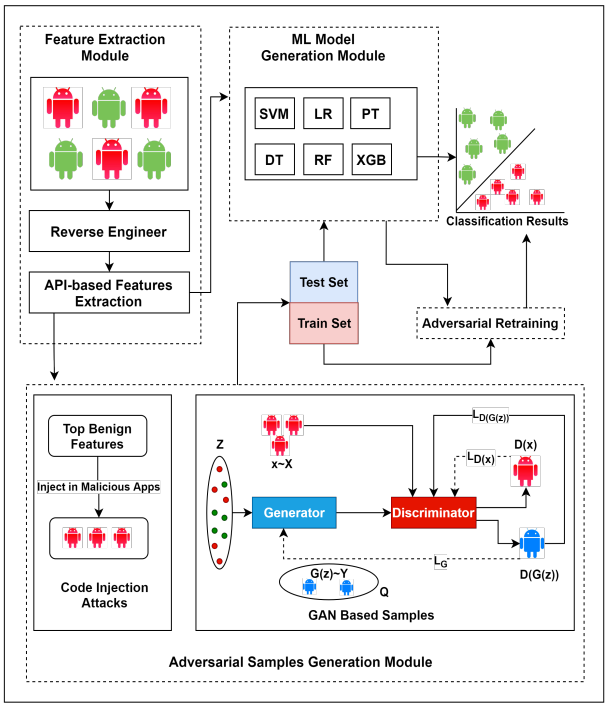

Fig. 1. Evasion Attacks Methodology

API-based features tend to be strong behaviour-based features for malware classification [1], [19]. A total of 315 unique API calls were found from all of the applications in the dataset. Furthermore, each application in the dataset is transformed into a feature vector of length 315. Each cell of the feature vector contains a binary value where 1 represents the presence of a specific feature and 0 represents its absence.

B. ML Models Segment

We used SVM, LR, PT, DT, RF and XGB classifiers on API-based features of APKs. All of the classifiers used default hyper-parameters setting (provided in sklearn 1.0.1 python library), whereas 10-folds were used for cross-validation. We distribute the dataset into 80% training set and 20% testing set for each iteration for cross-validation. In order to present the fragility of the Android malware classifiers, code injection and GAN-based adversarial evasion attacks will be applied. Subsequently, we will perform adversarial training to harden the security of models against such attacks.

C. Evasion Attacks Generator

In this section, we discuss our evasion attack strategies against pre-trained Android malware classifiers. The first step is to train the ML classifiers on an API-based dataset. Once the classifiers are trained, we apply code injection and GAN based evasion attacks on the classifiers.

1) Code Injection Attacks (CIA): To perform code injection attacks (CIA), we first find the top 20 most discriminating features from benign Android applications from the dataset

and then inject those one by one in the malicious applications. It has been observed that many API-based features are frequently used and are overlapping in both malicious and benign applications e.g. *StartActvity()*, *GetDeviceId()*, *GetActiveNetworkInfo()* etc. However, some features are highly discriminating in the sense of defining the class of Android applications, e.g. *sendTextMessage()* API is present in 1903 malicious applications in the dataset, whereas only 42 benign applications call this API.

Algorithm 1 shows the pseudo-code of features injection attack. The dataset of malicious Android apps *M* and top 20 features of benign class from Drebin F_{Top} are provided as input to the algorithm. Once the top features are identified, we look for those in the feature vectors of malicious apps. If a feature is missing, i.e. 0 in the malicious samples, we change it to 1 *(Algorithm 2, lines 1-4)*. The process of adding the features is carried out linearly, i.e. we mutate 1 top feature in all the malicious samples from 0 to 1 and test the samples on the model *(Algorithm 2, line 6)* to find out the evasion rate. Subsequently, the second top feature is mutated and then tested on the model and the same process is applied for the top 20 features.

Algorithm 1 CIA Algorithm

Input: $M = \{m_1, m_2, m_3 \ldots\ldots m_n\}$ and $F_{Top} = \{F_1, F_2, F_3 \ldots..F_{20}\}$
Output: E_{Rate}

```
for all i ∈ F do
    for all j ∈ M do
        if i ∈ j == 0 then
            j[F[i]] ← 1
        end if
        M_Evade ← j
    end for
    E_Rate ← Classifier(M_Evade)
end for
Return E_Rate
```

2) GAN Adversarial Examples Attacks (GAEA): A GAN is a combination of two neural networks, among which one is called generator ($\mathcal{G}$) and the other is known as discriminator ($\mathcal{D}$). ($\mathcal{G}$) generates the data and ($\mathcal{D}$) evaluates this generated data. Both these networks are connected in a way that the loss of $\mathcal{D}$ is fed back to $\mathcal{G}$ while $\mathcal{D}$'s weights are not updated so that $\mathcal{G}$ can try to follow the real data probability distribution more efficiently and fool the $\mathcal{D}$. In this work, the primitive version of GAN, also called vanilla GAN was used, to keep the experiments simplistic for estimating GANs potential for Android API based data generation. There is a further research gap for the exploration of a suitable GAN for the generation of Android API data. We leave this as future work. The generator model $\mathcal{G}$ in original/vanilla GAN can be represented as $\mathcal{G}$:z $\rightarrow$ $\mathcal{X}$ where z is the normal distribution from noise space and $\mathcal{X}$ is the real data distribution. The discriminator $\mathcal{D}$:$\mathcal{X}$ $\rightarrow$ [0,1] model is a classifier that outputs an estimate of probability how much the data coming from $\mathcal{G}$, is real or fake. The loss function of the combined model can be represented by Equation 1.

TABLE I
GAN CONFIGURATION

Parameter	Value
Network Type	Densely Connected Feed Forward
Number of Layers	$\mathcal{G}$: 5, $\mathcal{D}/\mathcal{C}$: 4
Input Layer Activations	$\mathcal{G}$: relu , $\mathcal{D}$: relu
Output Layer Activations	$\mathcal{G}$: sigmoid , $\mathcal{D}$: sigmoid
Batch Size	128
Multiplier(n)	128
Neurons in Input Layer	$\mathcal{G}$: 128 , $\mathcal{D}$: 128
Neurons in Layer 1	$\mathcal{G}$: $n \times 1$ = 128, $\mathcal{D}$: $n \times 2$ = 256
Neurons in Layer 2	$\mathcal{G}$: $n \times 2$ = 256, $\mathcal{D}$: $n \times 1$ = 128
Neurons in Layer 3	$\mathcal{G}$: $n \times 3$ = 384
Neurons in Output Layer	$\mathcal{G}$: 315, $\mathcal{D}/\mathcal{C}$: 1
Layer Regularization	$\mathcal{G}, \mathcal{D}$: $BatchNorm$
Optimizer	Adam (beta_1=0.5, beta_2=0.9)
Loss Function	binary cross entropy
Learning Rate	1e-5

$$\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{D}, \mathcal{G}) = \mathbb{E}_{x \sim p_{data}(x)}[\log \mathcal{D}(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - \mathcal{D}(\mathcal{G}(z)))] \quad (1)$$

Here, $\mathbb{E}$ stands for the probability estimation; x and z are the real and noise samples, respectively, while p_{data} and p_z represent the probability distributions of real and noise data. The goal in the mini-max game is to minimise the $\mathcal{G}$ loss in creating data similar to the real data. Since the generator can not control the loss of $\mathcal{D}$ on real data but it can maximise the loss of $\mathcal{D}$ on generated data $\mathcal{G}(z)$. The loss function of $\mathcal{G}$ is given by Equation 2.

$$J^{\mathcal{G}}(\mathcal{G}) = \mathbb{E}_{z \sim p_z(z)}[\log(\mathcal{D}(\mathcal{G}(z)))] \quad (2)$$

Table I shows the hyperparameter settings for the GAN model. It can be observed from this table that we used 'sigmoid' in the output layer of $\mathcal{G}$ due to the reason that we wanted to generate the API data in which the values need to be between 0 and 1.

We propose a GAN based methodology inspired by [20] that could mimic and generate the API based APK feature set. We propose the GAN evaluation by tweaking the classifier two-sample test (C2ST) [21] for $\mathcal{G}$ performance evaluation. The C2ST is a quantitative metric to compare two different samples of data. In other words, if we have samples real_APK_API data ($\mathcal{X}_m$) and GAN_APK_API data ($G(z)$), then we can assess if both samples have similar or the same probability distributions. The more the distributions overlap, the more is the chance that GAN_APK_API samples are realistic. The C2ST method has been shown in Algorithm 2. Here, $\mathcal{A}$ denotes the accuracy after splitting the input m samples from ($\mathcal{X}$) i.e. ($\mathcal{X}_m$) into 80% train set t_r and 20% test set t_s. The accuracy $\mathcal{A}$ is computed as per the Equation 4.

The GAN evaluation used in GAEA is different from C2ST in the evaluation parameter. The intuition is that the metric in C2ST, i.e. 'accuracy', should be replaced with the evasion

Algorithm 2 C2ST Algorithm

Input: $\mathcal{X}_m$ (real_APK_API samples), $G(z)$ (GAN_APK_API samples), Classifier

Output: $\mathcal{A}$(accuracy)

$t_r \leftarrow \mathcal{X}_m[0 : m(8/10)] \cup G(z)[0 : m(8/10)]$
$t_s \leftarrow \mathcal{X}_m[m(8/10) : m] \cup G(z)[m(8/10) : m]$
train ML_classifier on t_r
test ML_classifier on t_s
Return $\mathcal{A}$ = (TP + TN)/(TP + TN + FP + FN)

rate (e_{Rate}) if we want to reduce the false negatives in the classifier performance in post augmentation testing. The false negatives are the possible evasions that are already present in the test set, which the classifiers are not trained on. Hence, the C2ST has been tweaked so that the objective function becomes as given in the Equation 3. In Equation 3, $e_{Rate(\hat{a}rgmax)}$ is the evasion rate on D_{test} which is test set, n_{test} is the total number of samples in test set, z_i are the samples in test set, l_i are the labels, $f(z_i)$ is the conditional probability distribution $p(l_i = 1|z_i)$ and $\mathbb{I}$ is the indicator function. The intuition is that if a GAN_APK_API data is very close in probability distribution with a real_APK_API samples, then the evasion rate in Equation 3 should remain close to 100%. This means that the classifier was totally evaded, or the sample was misclassified as real_APK_API data. So if we use the evasion rate as the metric instead of accuracy, then we can better minimise the false negatives due to the reason that accuracy includes the value for false positives (FP) and true negatives (TN) given by Equation 4. Since our objective function is to minimise false negatives in generator evaluation so we must choose the epochs in which the evasion was the highest instead of accuracy being the lowest.

$$e_{Rate(\hat{a}rgmax)} = \frac{1}{n_{test}} \sum_{z_i, l_i \in D_{test}} \mathbb{I}[\mathbb{I}(f(z_i) > \frac{1}{2}) = l_i] \quad (3)$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (4)$$

However, in the evasion rate, we only have true positives (TP) and false negatives (FN) as given by Equation 5.

$$Evasion_{Rate} = \frac{FN}{TP + FN} \quad (5)$$

In Algorithm 3, first of all, we need to extract the malicious real_APK_API samples $\mathcal{X}_m$ from the preprocessed train set $\mathcal{T}$. Then we create GAN models and start training for 150 epochs in which for the batch size of 128, in each batch iteration, x_i is taken as a random batch from $\mathcal{X}_m$. We use the normal distribution of mean = 0 and standard deviation = 1 of the same size as of batch for noise input to $\mathcal{G}$. The $\mathcal{G}$ and $\mathcal{D}$ compute their gradients and update in backpropagation. After each epoch, we generate data G_{z_i} of size equal to $\mathcal{X}_m$ and add in a set U. Now, we can perform the proposed method to evaluate the performance of $\mathcal{G}$ in terms of evasion rate e_{Rate}. We perform 10-fold train-test split with 70-30 ratio

Algorithm 3 GAEA Algorithm

Input: $\mathcal{T}$ (preprocessed train set in csv format), batch_size, epochs, batches, Classifier

Output: E_{Rate}

$\mathcal{X}_m \sim \mathcal{T}$
Create $\mathcal{G}$ and $\mathcal{D}$ models
for $i \in epochs$ **do**
  for $j \in batches$ **do**
    $x_i \sim \mathcal{X}_m$
    $z_j \leftarrow \mathcal{N}_{\{mean=0,std=1,size=batch\_size\}}$
    $g_{z_j} \leftarrow \mathbb{E}_{z_i \sim p(z_j)}$
    $\theta_{\mathcal{D}j} \leftarrow \theta_{\mathcal{D}j} - \eta \nabla \theta_{\mathcal{D}j} \mathcal{L}(x_j)$
    $\theta_{\mathcal{D}j} \leftarrow \theta_{\mathcal{D}j} - \eta \nabla \theta_{\mathcal{D}j} \mathcal{L}(g_{z_j})$
    $\theta_{\mathcal{G}_j} \leftarrow \theta_{\mathcal{G}_j} - \eta \nabla \theta_{\mathcal{G}_j} \mathcal{L}(z_j)$
  end for
  $\theta_{\mathcal{G}i} \leftarrow \theta_{\mathcal{G}j}$
  $z_i \leftarrow \mathcal{N}_{\{mean=0,std=1,size=sizeof(\mathcal{X}_m)\}}$
  $G_{z_i} \leftarrow \mathbb{E}_{z_i \sim p(z)}$
  U = $\mathcal{X}_m \cup G_{z_i}$
  for $k \in 10$ **do**
    split_pointer = k
    $t_r \leftarrow$ 80% of U
    $t_s \leftarrow$ 20% of U
    train Classifier on t_r
    test Classifier on t_s
    compute e_{Rate}
  end for
  Compute $e_{Rate_{avg}}$
end for
$z \leftarrow \mathcal{N}_{\{mean=0,std=1,size=Normal-real\_APK\_APIsamples}$
$Gz_{\mathcal{I}_{argmax}(e_{Rate_{avg}})} \leftarrow \mathbb{E}_{z \sim p(z)}$
$E_{Rate} \leftarrow Classifier(Gz_{\mathcal{I}_{argmin}(E_{Rate})})$
Return E_{Rate}

and compute the average evasion rate $e_{Rate_{avg}}$. After the training is complete, we use the weights of the $\mathcal{G}$ for the epoch in which the value of $e_{Rate_{avg}}$ was maximum and generate GAN_APK_API data $Gz_{\mathcal{I}_{argmax}(e_{Rate})}$. The GAEA algorithm then outputs the E_{Rate} illustrated in Figure 4 the details of which will be mentioned in section III.

III. Experimental Results and Analysis

In this section, we evaluate the performance of different ML classifiers against code injection attacks (CIA) and GAN adversarial examples attacks (GAEA). Furthermore, we perform adversarial training of ML classifiers on CIA called CIA Adversarial Training AT or 'CIA AT' and GAEA Adversarial Training or 'GAEA AT' to improve the evasion detection of classical ML classifiers and evaluate against evasion attacks. We also perform GAEA on classifiers trained with CIA AT and CIA on classifiers trained with GAN AT. Finally, we perform evasion attacks on TrickDroid, a proposed adversarial training scheme on both CIA and GAEA based data and record the evasion rate in Figure 4. We use an API-based dataset which is composed of 5560 malicious and 5600 benign

TABLE II
CLASSIFICATION RESULTS

	Precision	Recall	F1-measure	Accuracy
SVM	0.898	0.841	0.868	0.876
Logistic regression	0.891	0.840	0.865	0.872
Perceptron	0.717	0.914	0.804	0.783
Decision Tree	0.924	0.870	0.896	0.902
Random forest	0.927	0.881	0.904	0.908
Xgboost	0.897	0.831	0.862	0.871

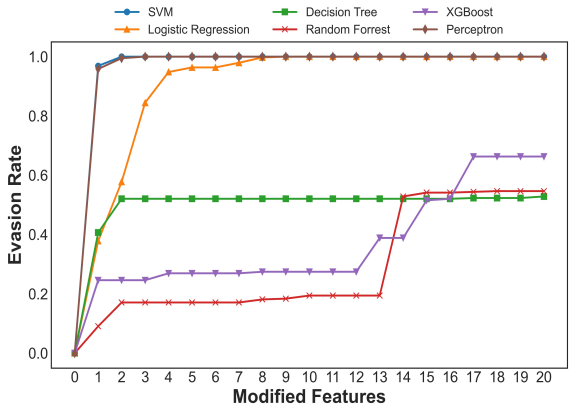

Fig. 2. Code Injection Attack

Android applications for the experiments. The experiments were performed on Dell G3 with 2.60GHz 6 core(s) processor, 16GB RAM and NVIDIA RTX 2060 GPU, running Windows 10.

In case of no adversarial attacks (NAT), we train the SVM, LR, PT, DT, RF and XGB classifiers on default hyper-parameters settings with 10-folds cross-validation with a distribution of 80% train set and 20% test set on each iteration. Table II presents the classification results obtained by the classifiers trained on API-based features. Amongst all the other classifiers, RF yields remarkable classification results with 90.8% accuracy. Figure 2 presents the results of CIA where the x-axis presents the number of features injected, and the y-axis represents the evasion rate. Consequently, linear classifiers SVM, LR and PT are affected the most with an evasion rate of 100%, which means all the adversarial samples in the test set were evaded. In comparison, DT was evaded the least with an evasion rate of 44.82. The evaluation of the CIA shows that linear classifiers are very fragile against the CIA.

The next attack we performed was the GAEA on NAT classifiers. As shown in Figure 3, similar to the CIA, in the case of GAEA, linear classifiers were affected the most with an evasion rate of more than 85% in all cases, whereas DT was least affected as compared to all the other classifiers with an evasion rate of 46.14%. As compared to CIA, GAEA have a slightly lower evasion rate with no classifier being evaded 100%. However, both of the attacks (CIA and GAEA) have proved to be significantly effective in evading pre-trained classifiers on the Android malware dataset.

As a countermeasure to mitigate the effects of evasion

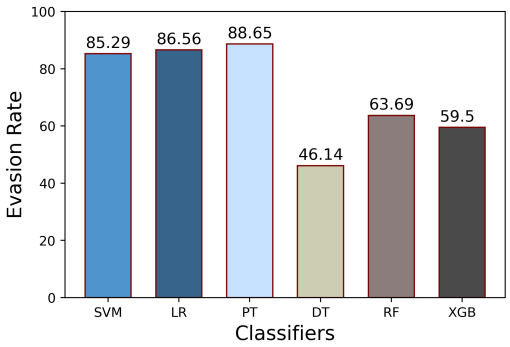

Fig. 3. Results of GAEA

attacks, we retrain classifiers on adversarial data and then evaluate those against evasion attacks. Firstly, we retrain the classifiers on code injection attacks (CIA AT) and perform the evaluation. As shown in Algorithm 1, to perform CIA, we inject the top features of benign Android applications in the malicious apps and evaluate those against pre-trained classifiers. We do so by first injecting the first top discriminating feature of benign apps into the malicious apps in all of the test sets and evaluating it against the classifiers. Furthermore, in addition to the first top feature, we inject the second top discriminating feature of the benign app in a malicious test set and perform the evaluation. The same process is applied till the injection of the top 20 benign features in the malicious test set. As discussed earlier, the CIA proved to be very effective to evade multiple ML classifiers. To perform retraining of classifiers on CIA, we generate an Oracle where for each malicious Android app, we added 20 new modified samples. The first sample has one top benign feature injected, the second sample has two benign top features injected and so on. Consequently, the size of the training set increased by 20 folds (i.e. 5560 to 111200). Although the size of the training set has dramatically increased, however, the CIA AT proved to be very effective. As a result of adversarial training of existing classifiers on CIA data (CIA AT in Figure 4), the most evaded classifier is XGB with only a 0.88% evasion rate.

In the next experiment, we perform GAEA Adversarial Training (GAEA AT) on the classifiers. We generate 5500 samples similar to the original malicious data using the method as mentioned in Section II. We augmented the GAEA data with the original dataset. As shown in Figure 4, classifiers trained on GAN adversarial examples (GAEA AT) perform remarkably well against GAEA with a worst-case of 12.53% evasion rate achieved in the case of PT trained on GAEA. All the other classifiers retrained on GAEA have an evasion rate of less than 10%. As compared to CIA AT, it is worth mentioning here that GAN based adversarial sample attacks were minimised by just retraining the classifiers on 5500 adversarial samples; however, to avoid CIA, we retrained classifiers on an Oracle of 111200 new samples as mentioned previously. Furthermore, we perform experiments by performing GAEA

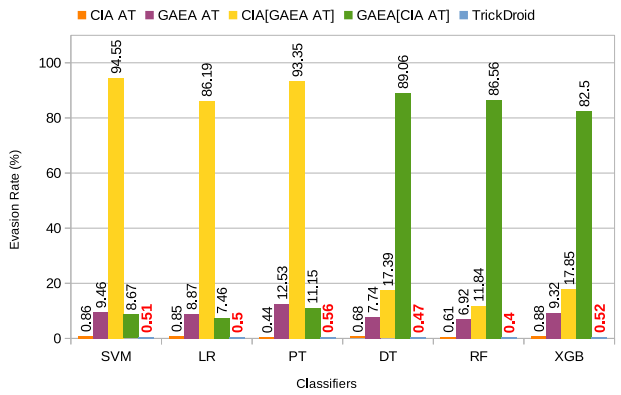

Fig. 4. Evasion Rate

on CIA AT and CIA on GAEA AT to cross-validate the efficacy of the two adversarial training CIA AT and GAEA AT on the classifiers. As shown in Figure 4, in case of performing CIA on GAN based adversarial trained classifiers (CIA[GAEA AT]), all the linear classifiers (SVM, LR and PT) have been evaded more than 85% whereas DT, XGB and RF perform very well with a worst-case evasion rate of 17.85% in case of XGB. Consequently, by applying GAEA on classifiers trained on code injection attacks (GAEA[CIA AT]), surprisingly, the results were opposite to CIA[GAEA AT]. As shown in Figure 4, in the case of GAEA[CIA AT], all the linear classifiers performed remarkably well with a worst-case evasion rate of 11.15% in the case of PT. Whereas DT, RF and XGBoost were evaded more than 82% in all cases. As a final countermeasure, we train classifiers on both CIA AT and GAEA AT and call this adversarial training as ***TrickDroid***. As shown in Figure 4 (highlighted in the red colour text), TrickDroid remarkably works well against both CIA[GAEA AT] and GAEA[CIA AT] with an evasion rate of no more than 0.51 in the worst case.

IV. Conclusion

The excessive use of Machine learning (ML) classifiers in Android malware detection demands a greater degree of inherent security due to the threats of adversarial evasion attacks. In this work, we highlight the fragility of classical ML classifiers against these types of attacks. After performing Oracle and GAN adversarial examples based attacks on different ML classifiers on a public Android dataset, we demonstrate an evasion rate of up to 100%. Our experiments reveal that the linear classifiers are less robust as compared to their ensemble counterparts both in Oracle and GAN based attacks. Furthermore, we present that despite adversarial training against one attack type, the classifiers are still vulnerable to other attacks. Hence, in order to further ruggedize the classifiers, we propose a Trickdroid, a cumulative adversarial training technique and demonstrate its efficacy with upto 99.46% evasion detection.

Acknowledgment

This work is supported by Northumbria's Academic Centre of Excellence in Cyber Security Research (ACE-CSR), and we are thankful for the support.

References

[1] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Make evasion harder: An intelligent android malware detection system." in *IJCAI*, 2018, pp. 5279–5283.

[2] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha-sly malware! scorpion a metagraph2vec based malware detection system," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 253–262.

[3] D. Li, Q. Li, Y. Ye, and S. Xu, "Sok: Arms race in adversarial malware detection," *arXiv preprint arXiv:2005.11671*, 2020.

[4] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, "Deceiving end-to-end deep learning malware detectors using adversarial examples," *arXiv preprint arXiv:1802.04528*, 2018.

[5] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *arXiv preprint arXiv:1606.04435*, 2016.

[6] H. Berger, C. Hajaj, and A. Dvir, "When the guard failed the droid: A case study of android malware," *arXiv preprint arXiv:2003.14123*, 2020.

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

[8] E. Aydogan and S. Sen, "Automatic generation of mobile malwares using genetic programming," in *European conference on the applications of evolutionary computation*. Springer, 2015, pp. 745–756.

[9] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016, pp. 365–376.

[10] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, and D. Clark, "Picking on the family: Disrupting android malware triage by forcing misclassification," *Expert Systems with Applications*, vol. 95, pp. 113–126, 2018.

[11] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 3, pp. 1–29, 2018.

[12] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan," *arXiv preprint arXiv:1702.05983*, 2017.

[13] X. Zhang, J. Wang, M. Sun, and Y. Feng, "Andropgan: An opcode gan for android malware obfuscations," in *International Conference on Machine Learning for Cyber Security*. Springer, 2020, pp. 12–25.

[14] H. Li, S. Zhou, W. Yuan, J. Li, and H. Leung, "Adversarial-example attacks toward android malware detection system," *IEEE Systems Journal*, vol. 14, no. 1, pp. 653–656, 2019.

[15] S. Jan, T. Ali, A. Alzahrani, and S. Musa, "Deep convolutional generative adversarial networks for intent-based dynamic behavior capture," *International Journal of Engineering & Technology*, vol. 7, no. 4.29, pp. 101–103, 2018.

[16] R. Taheri, R. Javidan, M. Shojafar, P. Vinod, and M. Conti, "Can machine learning model with static features be fooled: an adversarial machine learning approach," *Cluster Computing*, pp. 1–21, 2020.

[17] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[18] D. Li and Q. Li, "Adversarial deep ensemble: Evasion attacks and defenses for malware detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3886–3900, 2020.

[19] S. Wang, Z. Chen, X. Yu, D. Li, J. Ni, L.-A. Tang, J. Gui, Z. Li, H. Chen, and S. Y. Philip, "Heterogeneous graph matching networks for unknown malware detection." in *IJCAI*, 2019, pp. 3762–3770.

[20] R. H. Randhawa, N. Aslam, M. Alauthman, H. Rafiq, and F. Comeau, "Security hardening of botnet detectors using generative adversarial networks," *IEEE Access*, 2021.

[21] D. Lopez-Paz and M. Oquab, "Revisiting classifier two-sample tests," *arXiv preprint arXiv:1610.06545*, 2016.