Forensic Science International: Digital Investigation 44 (2023) 301511

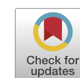
Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2023 EU - Selected papers of the Tenth Annual DFRWS Europe Conference

Adversarial superiority in android malware detection: Lessons from reinforcement learning based evasion attacks and defenses



Hemant Rathore [a, *], Adarsh Nandanwar [a], Sanjay K. Sahay [a], Mohit Sewak [b]

[a] *Dept. of CS & IS, Goa Campus, BITS Pilani, India*
[b] *Security & Compliance Research, Microsoft, India*

ARTICLE INFO

Article history:

Keywords:
Android
Adversarial robustness
Machine and deep learning
Malware detection
Reinforcement learning

ABSTRACT

Today, android smartphones are being used by billions of users and thus have become a lucrative target of malware designers. Therefore being one step ahead in this zero-sum game of malware detection between the anti-malware community and malware developers is more of a necessity than a desire. This work focuses on a proactive adversary-aware framework to develop adversarially superior android malware detection models. We first investigate the adversarial robustness of thirty-six distinct malware detection models constructed using two static features (permission and intent) and eighteen classification algorithms. We designed two Targeted Type-II Evasion Attacks (*TRPO-MalEAttack* and *PPO-MalEAttack*) based on reinforcement learning to exploit vulnerabilities in the above malware detection models. The attacks aim to add minimum perturbations in each malware application and convert it into an adversarial application that can fool the malware detection models. The TRPO-MalEAttack achieves an average fooling rate of 95.75% (with 2.02 mean perturbations), reducing the average accuracy from 86.01% to 49.11% in thirty-six malware detection models. On the other hand, The PPO-MalEAttack achieves a higher average fooling rate of 96.87% (with 2.08 mean perturbations), reducing the average accuracy from 86.01% to 48.65% in the same thirty-six detection models. We also develop a list of the *TEN* most vulnerable android permissions and intents that an adversary can use to generate more adversarial applications. Later, we propose a defense strategy (*MalVPatch*) to counter the adversarial attacks on malware detection models. The MalVPatch defense achieves higher detection accuracy along with a drastic improvement in the adversarial robustness of malware detection models. Finally, we conclude that investigating the adversarial robustness of models is necessary before their real-world deployment and helps achieve adversarial superiority in android malware detection.

© 2023 The Author(s). Published by Elsevier Ltd on behalf of DFRWS This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

In the last decade, smartphones have become a vital part of our daily lives and a dominant player in the communication & computing industry. The Global Digital Report (2022) suggests that the number of smartphone users has reached 5.31 billion, which is 67.1% of the world's population. The Android Operating System (OS) holds a monopoly in the mobile OS space with a market share of 70.01%. The android OS and applications (apps) hold a massive amount of users' personal and professional data, making it a very lucrative target for malware developers. According to AV-TEST (2022), the number of malware and PUA in the android ecosystem crossed 50 million in December 2021. The Zimperium Global Mobile Threat Report (2022) suggests that 77% of android apps/devices use at least one vulnerable encryption algorithm. The above facts suggest that any security breach in the android ecosystem can have catastrophic consequences on its user's data and privacy worldwide.

The anti-virus/anti-malware software products provide primary defense against malware attacks. However, the existing literature suggests that the current malware detection mechanisms like signature, heuristic, etc., cannot cope with new-age malware challenges. Therefore, anti-malware researchers have started exploring Machine Learning/Deep Learning (ML/DL) based android malware detection models that have shown encouraging results.

* Corresponding author.
E-mail address: hemantr@goa.bits-pilani.ac.in (H. Rathore).

https://doi.org/10.1016/j.fsidi.2023.301511
2666-2817/© 2023 The Author(s). Published by Elsevier Ltd on behalf of DFRWS This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Arora et al. (2019) showed that analysis of android permission pair usage could help in malware detection and achieved an accuracy of 95.44%. Zhu et al. (2020) proposed SEDMDriod that used a stacked ensemble of multi-layer perceptron and support vector machine to achieve an accuracy of 94.92% in malware detection. Zou et al. (2021) performed a graph-based analysis of sensitive API for android malware detection. Today, anti-virus/anti-malware companies have also started to integrate ML/DL based mechanisms for malware detection in their security products [1,2].

Recently researchers in other domains have found that ML/DL models are vulnerable to adversarial attacks. An adversary can intentionally add perturbations to the sample to fool the classification model. The first exhaustive study on this was performed by Goodfellow et al. (2015). They intentionally added small worst-case perturbations to create adversarial examples that can fool the image classification model. Since then, the researchers have started investigating adversarial robustness in computer vision, image classification, object detection, etc. problems and have shown promising results. However, similar studies on the adversarial robustness of android malware detection models are limited and minimal. The anti-virus companies have also reported (McAfee Mobile Threat Report (2022); Sophos Threat Report (2022)) that the threats posed by adversarial attacks on malware detection models are genuine and significant.

The *adversarial threat modeling* of android malware detection models can be defined based on the adversary's *objective*, *knowledge*, and *capability*. The *adversary's objective* can be to violate *integrity*, *privacy*, or *availability* of malware detection models. We perform *integrity* violations in malware applications and convert them into adversarial applications in this work. The *adversary's knowledge* about the target ecosystem can be defined based on three tuples: dataset, feature set, and classification algorithm. We assume a *white-box/perfect knowledge scenario* where the adversary has knowledge about all three tuples. The *adversary capability* can be defined based on the adversary's ability to perform read, modify or delete operations in the above three tuples. We perform *evasion attacks* by modifying test malware applications into adversarial applications that fool malware detection models by getting *forcefully misclassified* as benign.

Our work proposes a *proactive adversary-aware* framework that aims to construct *adversarially superior* android malware detection models. We first constructed *thirty-six* android malware detection models using two static features (android permission & android intent) and eighteen different classification algorithms from four distinct categories. Then we step into the *adversary's shoes* and design adversarial attacks based on Reinforcement Learning (RL) to exploit vulnerabilities in the above detection models. We propose two novel adversarial attacks: *Trust Region Policy Optimization-Malware Evasion Attack (TRPO-MalEAttack)* and *Proximal Policy Optimization-Malware Evasion Attack (PPO-MalEAttack)* against the detection models. The above attacks are *Targeted Type-II (False-Negative) Evasion Attacks* for the *white-box scenario* that convert malware applications into adversarial applications that can *fool* the malware detection models. The aim is to add *minimum perturbations* in each malware application so that it is forcibly misclassified as benign by the detection models. The attack agents also ensure that perturbations do not break the modified applications' structural, syntactical, functional, and behavioral integrity. The attack agents aim to convert all the malware applications into adversarial applications and thus drastically reduce the performance of detection models. Later, we analyzed the exposed vulnerabilities and used them in defense strategy to counter adversarial attacks. We proposed the *Malware Vulnerability Patch (MalVPatch)* based on adversarial retraining of detection models. The performance of MalVPatch defense is examined using improvement in detection accuracy and adversarial robustness of malware detection models. Finally, we made the following contributions.

- **Generalizability:** We develop thirty-six distinct android malware detection models using ML/DL. These models are constructed using two features (android *permission* & android *intent*) and eighteen classification algorithms derived from four categories (*Standard*, *Bagging*, *Boosting*, & *Deep Neural Network*). The thirty-six malware detection models achieve an average accuracy of 86.01%. (refer to section: 4.1)
- **RL Attack Policy:** We design two Targeted Type-II (False-Negative) Evasion Attacks (*TRPO-MalEAttack* and *PPO-MalEAttack*) using reinforcement learning against the above thirty-six malware detection models. The attack agents are designed to add minimum perturbations in malware applications to transform them into adversarial applications that are misclassified as benign and thus fool the malware detection models. (refer to section: 2.2)
- **Adversarial Attack:** The *TRPO-MalEAttack* achieves an average fooling rate of 95.75% with 2.02 mean perturbations and reduces the average accuracy from 86.01% to 49.11% for the thirty-six detection models. Similarly, *PPO-MalEAttack* achieves an average fooling rate of 96.87% with 2.08 mean perturbations and reduces the average accuracy from 86.01% to 48.65% for the thirty-six detection models. We also list the *TEN* most vulnerable android permissions and intents that can easily force misclassifications in detection models. (refer to section: 4.2)
- **Adversarial Defense:** We propose a defense strategy (*MalVPatch*) based on adversarial retraining to counter the evasion attacks on malware detection models. The MalVPatch defense improves the average detection accuracy from 86.01% to 89.97% for thirty-six TRPO-MalEAttack models and 86.01% to 90.02% for thirty-six PPO-MalEAttack models. (refer to section: 4.3)
- **Adversarial Robustness:** We reattack the MalVPatch models using the same evasion attacks to investigate their adversarial robustness. The average fooling rate of *MalVPatch* models reduces to 2.49% against TRPO-MalEAttack and 3.77% against PPO-MalEAttack that leads to adversarially superior MalVPatch detection models. (refer to section: 4.3.2)

The rest of the paper is structured as follows: *Section-2* describes the adversarial attacks and defense policies. *Section-3* explains the experimental setup. *Section-4* contains the experimental results. *Section-5* discusses the related work, and finally, *Section-6* concludes the paper.

2. Adversarial attack and defense

This section first explains the proposed framework, followed by adversarial attack strategies (TRPO-MalEAttack and PPO-MalEAttack) and defense strategy (MalVPatch).

2.1. Proposed framework

Fig. 1 provides a visual summary of the proposed framework for developing adversarially superior android malware detection models. The first step is **Data Collection** which involves gathering

[1] https://www.avast.com/en-in/technology/ai-and-machine-learning.
[2] https://www.mcafee.com/enterprise/en-us/assets/white-papers/wp-advanced-analytics-machine-learning.pdf.

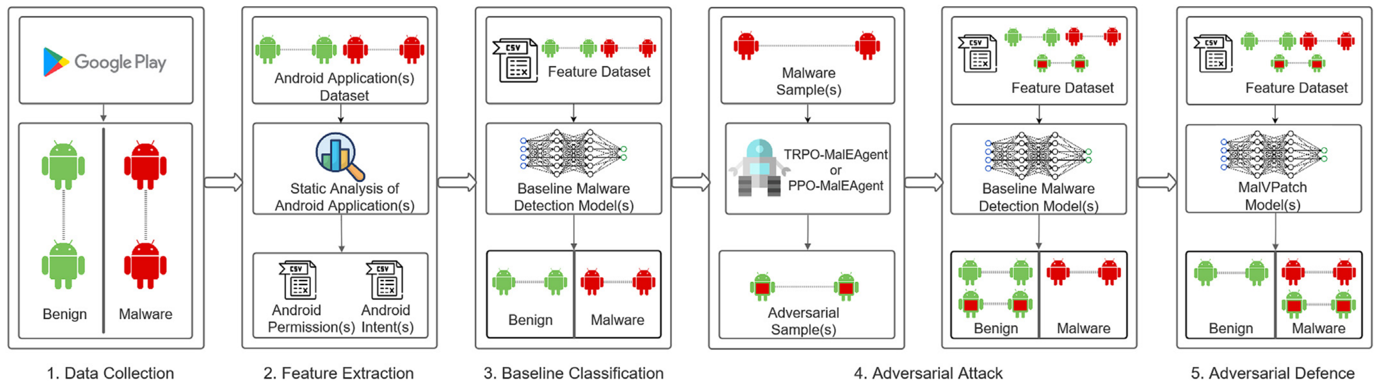


Fig. 1. Overview of the proposed framework to develop adversarially superior android malware detection models.

android applications (malware and benign) for the dataset. The second step is **Feature Extraction** which involves processing the android applications into a format that can be used to train ML/DL based malware detection models. We perform static analysis of applications to extract two features: android permission and android intent. The **Baseline Classification** step involves the construction of android malware detection models using the above extracted features. We constructed thirty-six distinct baseline malware detection models (based on eighteen classification algorithms and two features) and evaluated their performance. The fourth step is developing **Adversarial Attack**, which exploits vulnerabilities in the above detection models and investigates their adversarial robustness. We designed two attack strategies (*TRPO-MalEAttack* and *PPO-MalEAttack*) based on reinforcement learning, which perform targeted type-II (false-negative) evasion attacks by converting malware applications into adversarial applications and fooling the malware detection models. The last stage is developing **Adversarial Defense** *(MalVPatch)* to counter the evasion attacks and finally develop the adversarially superior android malware detection models.

2.2. Adversarial attack (MalEAttack)

We propose the Malware Evasion Attack (MalEAttack) based on the RL framework against android malware detection models. The MalEAttack is a Targeted False-Negative Evasion Attack in a white-box scenario against various detection models. The goal of the MalEAttack is to add perturbation(s) in malicious applications (target class) and convert them into adversarial applications that fool the malware detection models. The MalEAttack is optimized to achieve a high conversion rate (fooling rate) with minimum perturbations as lesser modifications are easier to perform in real-world applications and attract lesser suspicion. The MalEAttack also ensures that changes do not break the applications' structural, syntactical, functional, and behavioral integrity.

The RL framework for MalEAttack is explained as follows. The android applications are first represented using features F and the corresponding feature vector v. We used android permission/intent as features where $|F|$ is the number of permission/intent in the feature vector. The $v[i]$ is set to 1 iff the ith feature (permission/intent) is used by the application otherwise $v[i]$ is set to 0. Here, the vector v also represents the state of the MalEAttack RL environment. The set of all possible permutations of v makes up the multibinary observation space O. The Malware Evasion Attack Agent (MalEAgent) chooses an action $a \in A$, where $A \subseteq F$ is a finite set of actions representing a discrete action space. Each action a maps to an index in v using the mapping m. When the agent takes action a, v is modified to reach a new state v' using equation (1). The next state v' will have the value set to 1 at the index corresponding to the action selected by the agent. The values at all other indexes remain the same as the previous state v. This ensures the functional integrity of the new state since features are never removed.

$$v'[i] = \begin{cases} 1, & \text{if } i = m[a] \\ v[i], & \text{otherwise} \end{cases} \tag{1}$$

The environment returns a reward corresponding to the action chosen by the MalEAgent, which is calculated using a reward function. The reward function used in the paper is given by equation (2) where *score* is the probability estimate of the current state for the *benign* class calculated by the detection model *cls* being attacked. The range of the reward is a floating-point value in the range $[-1, 1]$.

$$reward_{cls,v,v'} = score_{cls,v'} - score_{cls,v} \tag{2}$$

Algorithm 1 describes the steps for converting malware applications into adversarial applications. The attack proceeds in two phases: exploration and exploitation. Line 1–2 creates an RL environment and the MalEAgent. The exploration phase is next in line 4, after which lines 8–20 are the exploitation phase.

2.2.1. Exploration

The RL framework for MalEAttack is illustrated in Fig. 2. The RL agent plays a game-like scenario where it takes specific actions to win the game. The agent's role is to observe the current state (malware application), perform an action (i.e., adding a permission/intent) from a set of options, and receive a reward for it. The agent then uses this reward value to modify its internal weights to improve its decision-making process and optimize the actions to reach the goal. The agent iteratively goes through all the malware applications in the training dataset X and plays the game multiple times, each time with a different malicious application. Each game

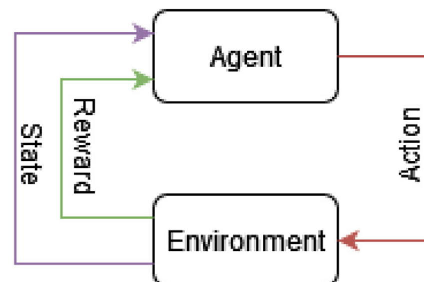


Fig. 2. Reinforcement learning framework.

iteration is known as an episode. The goal of the agent is to maximize the reward in each episode. The length of an episode is fixed as the parameter *ep_len*. A new episode begins when the environment is *reset*. On resetting, all the variables are reset to default values, with the state *v* being set to the next malware application in the dataset.

Algorithm 1. Algorithm for MalEAttack

Input:
model: malware detection model under attack
X: training data
A: action space
O: observation space
ep_len: episode length
m: action to index map
max_pert: maximum number of perturbations limit
attack_algo: attack algo (TRPO or PPO)
Function:
get_env: creates an RL environment with specified configuration
get_agent: creates the MalEAgent based on specified algorithm
get_action: returns action suggested by the MalEAgent based on state
learn: trains the agent on the dataset X
reset: resets the env and sets its variables to default value
predict: true when model classifies env state as malware
step: takes action a on env
append: adds element at the end of the list
Output:
M′: set of successfully modified adversarial applications
added_perturbations: list of added perturbations

1: $env \leftarrow get_env(model, X, A, O, ep_len, m, max_pert)$
2: $agent \leftarrow get_agent(attack_algo, env)$
3: # Exploration
4: $agent \leftarrow learn(agent, X)$
5: # Exploitation
6: $v \leftarrow reset(env)$
7: $M' \leftarrow []$
8: **for** $i = 1$ to $|X|$ **do**
9:     **if** $(predict(env))$ **then**
10:         $done \leftarrow False$
11:         **while** $done == False$ **do**
12:             $a \leftarrow get_action(agent, v)$
13:             $v, reward, done, success \leftarrow step(env, a)$
14:         **end while**
15:         **if** $(success)$ **then**
16:             $append(M', v)$
17:         **end if**
18:     **end if**
19:     $v \leftarrow reset(env)$
20: **end for**
21: return $(M', added_perturbations)$

The RL agent uses an algorithm (*attack_algo*) to learn optimal actions to reach the goal state. We use the Trust Region Policy Optimization (TRPO) algorithm in TRPO-MalEAttack and the Proximal Policy Optimization (PPO) algorithm in PPO-MalEAttack for this purpose.

2.2.2. TRPO-MalEAttack

The proposed TRPO-MalEAttack uses the *TRPO algorithm* to convert malware applications into adversarial applications to fool malware detection models. The TRPO is an RL algorithm proposed by Schulman et al. (2015) that works with both discrete and continuous action space. The proposed TRPO-MalEAttack incorporates the TRPO in the MalEAgent on discrete action space. The TRPO-MalEAgent uses an on-policy algorithm similar to natural policy gradient methods. However, unlike natural policy gradient methods, we use the $KL - Divergence$ constraint while updating the policy parameters. In normal policy gradient methods, the performance can have a huge impact even due to a change in policy parameters that might look small to us. This makes taking giant steps very risky. The TRPO-MalEAgent avoids this problem by using the constraints. It covers the largest distance within the constraint to ensure that the policy is not changed drastically. This ensures that the performance does not collapse due to a single wrong step.

2.2.3. PPO-MalEAttack

The PPO-MalEAttack uses the *PPO algorithm* on discrete action space to learn the adversarial attack strategy. The PPO is a family of policy gradient methods introduced by Schulman et al. (2017). These new methods share some benefits like stability and reliability of TRPO while having a simpler implementation. The PPO-MalEAgent works on a similar idea as TRPO-MalEAgent on taking the biggest step within a constraint to avoid a huge change in policy, preventing a collapse of performance. This also uses an on-policy method. However, unlike TRPO-MalEAgent, which uses complex methods, PPO-MalEAgent uses a simpler implementation involving only first-order optimization methods. The PPO algorithm has two variants: Adaptive KL Penalty Coefficient and Clipped Surrogate Objective without having any constraint. The PPO-MalEAgent uses *PPO Clipped Surrogate Objective* approach, where the KL constraint objective is replaced by directly clipping the objective function.

2.2.3.1. Exploitation.

In the exploitation phase, the MalEAgent uses its knowledge learned from the exploration phase (final weights). The MalEAgent goes through each malware application in the test dataset, and tries to convert it into an adversarial application using algorithm 1 (steps 9–18). The MalEAgent suggested actions are performed until one of the following conditions is met.

1. Malware application successfully converted into adversarial malware application. In this case, the episode ends early and the MalEAttack is successful.
2. MalEAttack agent is unsuccessful because the episode length has reached its set limit.

2.3. Adversarial defense strategy (MalVPatch)

We propose *Malware Vulnerability Patch (MalVPatch)* as a defense strategy to improve the adversarial robustness of android malware detection models. The method involves adversarially retraining the malware detection models and patching the blind spots exposed by various adversarial attacks. Grosse et al. (2016) compared various defense approaches and found that adversarial retraining achieved the best results against adversarial attacks. Taheri et al. (2020) also found that adversarial retraining is better than the generative adversarial network defense strategy in the malware detection domain. Rathore et al. (2021c) explored three defense approaches and found adversarial retraining achieved better results. The *adversarial retraining* defense is a highly generalizable approach and can be used with almost all the classification algorithms. On the other hand, *distillation* based defense transfers knowledge from a bigger model to a small model, which might increase bias in classification models. Another limitation of distillation is that it can only be used with deep neural network models and cannot be used with machine learning, bagging, or boosting models. Similarly, *feature reduction* defense might suffer from the curse of dimensionality, and finding relevant features using mutual information, variance threshold, etc., adds additional overhead. Thus we decided to explore *adversarial retraining* based MalVPatch defense in this work. The first step in MalVPatch is to obtain the adversarial applications using the MalEAgent attacks. We then construct a new training dataset by importing the original dataset along with their class labels and then adding the adversarial applications with their class label set to *malware*. The addition of adversarial applications may result in a significant imbalance in the class distribution of the new dataset. Thus we perform *class balancing* using oversampling in our experiments.

3. Experimental setup

This section first discusses the dataset, feature extraction, and classification algorithms, followed by the performance metrics and experimentation platform used in the paper.

3.1. Dataset (malware and benign)

The first stage in the proposed framework (Fig. 1) is to construct a well-labeled dataset of android applications containing malware and benign samples. Our dataset contains android applications downloaded from Google Play Store[3] for benign samples and the Drebin dataset[4] for malicious samples. We downloaded 8000 + android applications from Google Play Store and verified them using the services of VirusTotal[5]. The downloaded application is labeled *benign* only if all the antiviruses from VirusTotal declare it non-malicious and the rest of the samples are discarded. The final *benign dataset* contains 5721 android applications. On the other hand, we used the benchmark Drebin dataset for the *malware dataset* that contains 5553 malicious applications for our experiments.

3.2. Feature extraction

The second stage in the proposed framework (Fig. 1) is feature extraction from android applications collected in the dataset. We extracted two distinct features: *android permission* and *android intent* to construct malware detection models. We used android permission since it is responsible for providing user security by protecting data and action accesses in the android ecosystem. We also used android intent which provides APIs for various operations, again being used as a security mechanism. The android applications in the dataset are decompiled using the reverse engineering tool *(Apktool*[6]). Then we created exhaustive lists of permission[7] and intent[8] using official android documentation. The permission and intent lists contain 195 permissions and 273 intents. We developed a parser that scans each android application and logs its permission usage to create a corresponding **android permission** feature vector. If an application uses a particular permission, then the value at corresponding index in the feature vector will be set to 1; otherwise, 0. We developed a similar parser for intent as well and performed the same exercise to generate **android intents** feature vector.

3.3. Classification algorithms

We used *eighteen* distinct classification algorithms derived from *four* different categories to demonstrate the **generalizability** of the proposed approaches (Malware Detection Model, TRPO-MalEAttack, PPO-MalEAttack, and MalVPatch). Table 1 shows the four categories and the corresponding classification algorithms.

The paper (Rathore et al. (2020) & Rathore et al. (2021b)) explains a detailed description of classification algorithms, their architecture & parameters, and the corresponding android malware detection models.

[3] https://play.google.com/store?hl=en.
[4] https://www.sec.cs.tu-bs.de/danarp/drebin/.
[5] https://www.virustotal.com/.
[6] https://ibotpeaches.github.io/Apktool/.
[7] https://developer.android.com/guide/topics/permissions.
[8] https://developer.android.com/reference/android/content/Intent.

3.4. Performance metrics

We use the following performance metrics to evaluate and compare our results in this paper.

- **True Positive (TP)**: Number of malicious apps that are correctly classified as malware by the model.
- **True Negative (TN)**: Number of benign apps that are correctly classified as benign by the model.
- **False Positive (FP)**: Number of benign apps that are misclassified as malware by the detection model.
- **False Negative (FN)**: Number of malicious apps that are misclassified as benign by the detection model.
- **Accuracy (Acc)**: The percentage ratio of apps correctly classified by the malware detection model to the total number of apps.
- **Fooling Rate (FR)**: The percentage ratio of malware apps successfully converted into adversarial apps using MalEAttack to the number of correctly classified malware apps (before the attack) by the detection model. The adversary's goal (MalEAttack) is to maximize the fooling rate. Here, M is the set of malware apps, and M' is the set of adversarial apps.

$$FR = \frac{|M'|}{|M| - FN} \times 100 \quad (3)$$

- **Perturbation Count (PC)**: Minimum number of perturbation(s) performed to convert a malware app into an adversarial app. The adversary's goal (MalEAttack) is to minimize PC for each adversarial app.
- **Perturbation Application Percentage (PAP)**: PAP of a perturbation is the percentage ratio of the number of adversarial apps in which that perturbation (permission/intent) was added during the MalEAttack to the total number of adversarial apps.
- **Perturbation Frequency Percentage (PFP)**: It is the percentage ratio of the frequency of a particular perturbation (permission/intent) to the total number of perturbations during the MalEAttack. The Goal is to identify common perturbations to construct vulnerability lists (permission and intent).

3.5. Experimentation platform

The *Google Colab Pro* platform is used for all the experiments, namely baseline malware detection models, adversarial attacks (TRPO-MalEAttacks & PPO-MalEAttack), and defense (MalVPatch). The code is written in python language using *scikit-learn*, *tensorflow2*, and *open AI gym*. The dataset is divided into train and test sets in the ratio of 70:30 for training and evaluation, respectively. All the adversarial applications are recompiled using *Apktool* and validated for structural, syntactical, functional, and behavioral integrity. The code will be made open-source on *GitHub* for the wider community *after acceptance*.

4. Experimental results

This section presents the experimental results with relevant figures. First, we will discuss the results obtained by baseline android malware detection models, followed by adversarial attacks (TRPO-MalEAttack and PPO-MalEAttack) and then adversarial defense (MalVPatch).

Table 1
Categorization of classification algorithms to construct malware detection models.

Standard Classification Algorithm(s)	Bagging based Classification Algorithm(s)	Boosting based Classification Algorithm(s)	Deep Neural Networks based Classification Algorithm(s)
Logistic Regression (LR)	Random Forest (RF)	Adaptive Boosting (AB)	Deep Neural Network 0 Hidden Layer (DNN0L)
Support Vector Machine (SVM)	Bootstrap Aggregating Decision Tree (BADT)	Gradient Boosted Regression Trees (GBRT)	Deep Neural Network 1 Hidden Layer (DNN1L)
Kernel Support Vector Machine (KSVM)	Bootstrap Aggregating Logistic Regression (BALR)	Extreme Gradient Boosting (EGB)	Deep Neural Network 2 Hidden Layer (DNN2L)
Decision Tree (DT)	Bootstrap Aggregating Kernel Support Vector Machine (BAKSVM)	Light Gradient Boosting Machine (LGBM)	Deep Neural Network 3 Hidden Layer (DNN3L)
		Cat Boost (CB)	Deep Neural Network 4 Hidden Layer (DNN4L)

4.1. Baseline android malware detection models

In the third stage in the proposed framework (Fig. 1), we train thirty-six different baseline android malware detection models using eighteen distinct classification algorithms and two static features. The performance of these baseline models is evaluated using accuracy, AUC, etc. Fig. 5(a) shows the accuracy (blue bar) of all the eighteen baseline android malware detection models (permission), and Fig. 5(b) (blue bar) shows the same for the intent feature. The average accuracy obtained across eighteen **malware detection models (permission)** is 93.55%. Here, the highest accuracy is obtained by DNN4L (95.27%) and the lowest by AB (91.22%). The DNN3L and RF also achieved high accuracies of 95.18% and 95.12%, respectively. Also, the average AUC score of 0.94 is achieved for eighteen malware detection models (permission), with the highest being for DNN4L (0.95) and the lowest for AB (0.91). On the other hand, an average accuracy of 78.47% was obtained for eighteen **malware detection models (intent)**. Here, the highest accuracy is obtained by CB (79.37%) and the lowest by SVM (77.27%). The RF and DNN4L also achieved high accuracies of 79.22% and 79.19%, respectively. Also, the average AUC score of 0.79 is achieved for eighteen malware detection models (intent), with the highest being for CB (0.79) and the lowest for AB (0.77). The above result also shows that malware detection models based on permission achieved better performance compared to the intent counterparts.

The above baseline malware detection models achieved similar performance compared to the state-of-the-art models discussed in the literature. For the Drebin permission dataset, Li et al. (2018), Arora et al. (2019), and Khariwal et al. (2020) achieved an accuracy of 91.34%, 95.44% and 94.73%, respectively. On the other hand, Sewak et al. (2020) achieved 77.2% accuracy and 0.81 AUC for the Drebin intent dataset.

4.2. Adversarial attack on malware detection models

The fourth stage in the proposed framework (Fig. 1) is to perform adversarial attacks (MalEAttack) on the baseline malware detection models constructed in the previous stage. The attack's goal (TRPO-MalEAttack & PPO-MalEAttack) is to convert the maximum number of malware applications into adversarial applications by adding minimum perturbations such that they are intentionally misclassified by malware detection models. The MalEAgent allows a maximum of six perturbations against malware detection models (permission), whereas a maximum of three perturbations for detection models (intent). The agents also ensure that these perturbations do not violate the structural, functional, and syntactical integrity of the modified malware applications. We used performance metrics like fooling rate, accuracy change, AUC change, and the number of perturbations to evaluate the impact of the adversarial attacks on malware detection models and draw comparisons.

4.2.1. Fooling rate @ MalEAttacks

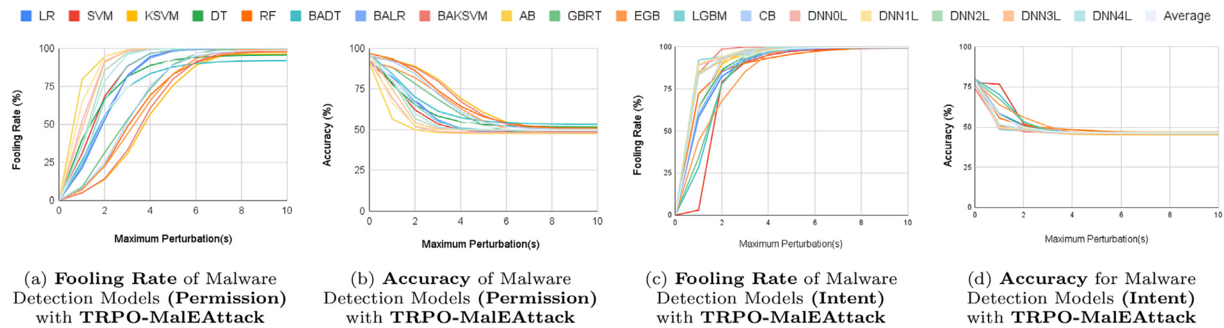
Fig. 3(a) and (c) show the fooling rate achieved by **TRPO-MalEAttack** against different android malware detection models (permission and intent). The TRPO-MalEAgent against eighteen malware detection models (permission) achieved an average fooling rate of 96.49% with 2.58 mean perturbations and a maximum limit of six perturbations. AB is the weakest model with a 100% fooling rate, whereas KSVM has the lowest fooling rate of 88.75%. On the other hand, the TRPO-MalEAgent against eighteen detection models (intent) achieved an average fooling rate of 95.01% with 1.46 mean perturbations and a maximum limit of three perturbations. The TRPO agent attains the highest and lowest fooling rate with BAKSVM (99.97%) and EGB (85.27%).

Fig. 4(a) and (c) show the fooling rate achieved by **PPO-MalEAttack** against different android malware detection models (permission and intent). The PPO-MalEAgent against eighteen detection models (permission) achieved an average fooling rate of 97.80% with 2.59 mean perturbations and a maximum limit of six perturbations. Here, the PPO agent achieved the highest fooling rate of 99.98% against DNN3L and DNN4L, whereas the lowest of 91.73% is attained against KSVM. On the other hand, the PPO-MalEAgent against eighteen detection models (intent) has an average fooling rate of 95.95% with 1.59 mean perturbations and a maximum limit of three perturbations. The PPO agent achieved the highest and lowest fooling rates with BAKSVM (99.97%) and EGB (83.70%).

The figures show a steep increase in the fooling rate for all the malware detection models (permission/intent) with the increase in the number of perturbations by MalEAgents. Also, the PPO-MalEAttack achieved a slightly higher fooling rate as compared to TRPO-MalEAttack. Finally, we can conclude that neither permission nor intent based malware detection models are resilient to the MalEAttacks.

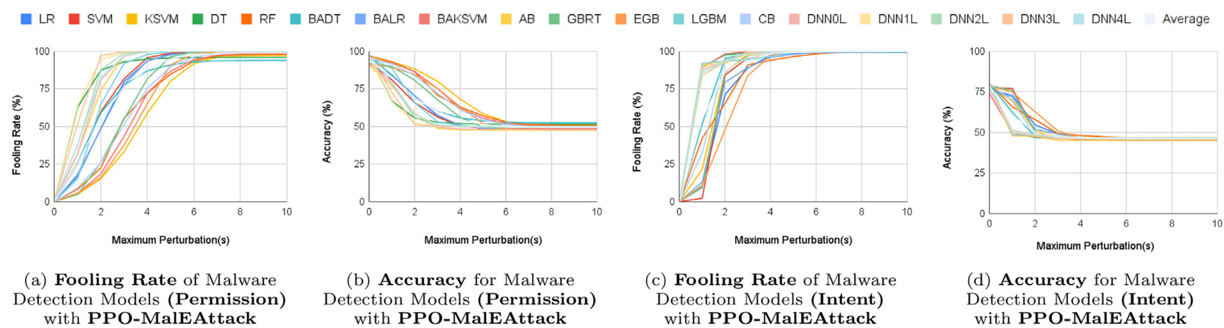
4.2.2. Accuracy reduction @ MalEAttacks

The MalEAttacks force a massive number of intentional misclassifications in all the baseline android malware detection models (permission/intent). Fig. 3(b) and (d) show the performance of different detection models (permission and intent) after **TRPO-MalEAttack**. The TRPO-MalEAttack (with a maximum of six perturbations) reduces the average accuracy of eighteen detection models (permission) from 93.55% to 50.61%. The maximum and minimum accuracy in detection models (permission) after TRPO-MalEAttack is shown by KSVM (54.55%) and AB (47.63%). On the other hand, TRPO-MalEAttack (with a maximum of three perturbations) reduces the average accuracy of eighteen detection models



(a) **Fooling Rate** of Malware Detection Models **(Permission)** with **TRPO-MalEAttack**
(b) **Accuracy** of Malware Detection Models **(Permission)** with **TRPO-MalEAttack**
(c) **Fooling Rate** of Malware Detection Models **(Intent)** with **TRPO-MalEAttack**
(d) **Accuracy** for Malware Detection Models **(Intent)** with **TRPO-MalEAttack**

Fig. 3. Performance of **TRPO-MalEAttack** against different Android Malware Detection Models (Permission/Intent).



(a) **Fooling Rate** of Malware Detection Models **(Permission)** with **PPO-MalEAttack**
(b) **Accuracy** for Malware Detection Models **(Permission)** with **PPO-MalEAttack**
(c) **Fooling Rate** of Malware Detection Models **(Intent)** with **PPO-MalEAttack**
(d) **Accuracy** for Malware Detection Models **(Intent)** with **PPO-MalEAttack**

Fig. 4. Performance of **PPO-MalEAttack** against different Android Malware Detection Models (Permission/Intent).

(intent) from 78.47% to 47.60%. The maximum and minimum accuracy in detection models (intent) after TRPO-MalEAttack is shown by EGB (50.37%) and DNN0L (46.35%).

Fig. 4(b) and (d) show the performance of different malware detection models (permission and intent) after **PPO-MalEAttack**. The PPO-MalEAttack (with a maximum of six perturbations) reduces the average accuracy of eighteen detection models (permission) from 93.55% to 50.02%, with maximum and minimum accuracy shown by KSVM (53.18%) and AB (47.64%). On the other hand, PPO-MalEAttack (with a maximum of three perturbations) reduces the average accuracy of eighteen detection models (intent) from 78.47% to 47.28%. The maximum and minimum accuracy is shown by EGB (50.89%) and AB (45.08%).

The above results suggest that intent-based malware detection models are more adversarially robust and require more perturbations for a similar accuracy reduction than permission-based models. As expected, a higher fooling rate with MalEAttacks results in a more significant reduction in the accuracy of detection models.

4.2.3. Vulnerable permissions and intents

We calculated **Perturbation Application Percentage** (PAP) and **Perturbation Frequency Percentage** (PFP) to understand the list of perturbations added by MalEAgents during the attack on android malware detection models. Table 2 lists the top ten permissions and top ten intents added by TRPO-MalEAgent and PPO-MalEAgent during the attacks on detection models.

The **TRPO-MalEAgent** added *android.permission.READ_CALL_LOG* in 59.91% of adversarial samples which account for 23.13% of all the permission perturbations. It was followed by *android.permission.USE_CREDENTIALS*. Similarly, TRPO-MalEAgent added *android.intent action.MY_PACKAGE_REPLACED* in 58.79% of adversarial samples which constituted 39.92% of all intent perturbations. The next most frequently used intents is *android.intent.action.MEDIA_BUTTON*.

The **PPO-MalEAgent** added *android.permission.READ_CALL_LOG* in 56.86% of adversarial samples which account for 22.04% of all the permission perturbations. It was again followed by *android.permission.USE_CREDENTIALS*. On the other hand, PPO-MalEAgent added *android.intent action.MY_PACKAGE_REPLACED* in 63.54% of adversarial samples, which constituted 40.04% of all intent perturbations. The next most frequently used intents is *android.intent.action.MEDIA_BUTTON*.

Some permissions/intents are used most of the time by adversaries (MalEAttack) to fool the malware detection models. Table 2 shows that the top 9 out of **10 permissions** are common between TRPO-MalEAttack and PPO-MalEAttack. On the other hand, all top **10 intents** are common for TRPO-MalEAttack and PPO-MalEAttack but in a slightly different order. This also shows that the above permission(s)/intent(s) are **highly vulnerable** and can be used by any adversary to attack detection models.

4.3. Adversarial defense strategy

In the fifth and the last stage in the proposed framework (Fig. 1), we proposed **MalVPatch** as the adversarial defense to improve the performance of malware detection models. The defense strategy involves gathering adversarial samples during the MalEAttacks and using them to adversarially retrain the detection models. The impact of defense strategy (a.k.a MalVPatch models) is measured using two factors. The first is improvement in malware detection performance (accuracy, AUC, etc.). The second is improvement in the adversarial robustness.

4.3.1. Detection performance @ MalVPatch models

Fig. 5(a) shows the performance of malware detection models (permission) after applying the defense strategy **MalVPatch on TRPO-MalEAttack** models. We observe an improvement in the accuracy of MalVPatch malware detection models (yellow bar) over baseline models (blue bar). The average accuracy of eighteen MalVPatch models (permission) improved by 2.02% from baseline

Table 2
Top *TEN* most vulnerable permissions and intents during the TRPO-MalEAttack and PPO-MalEAttack

	TRPO-MalEAttack			**PPO-MalEAttack**		
	Name	**Perturbation Frequency Percentage (PFP)**	**Perturbation Application Percentage (PAP)**	**Name**	**Perturbation Frequency Percentage (PFP)**	**Perturbation Application Percentage (PAP)**
Permission (Maximum 6 perturbations)	android.permission.READ_CALL_LOG	23.133	59.908	android.permission.READ_CALL_LOG	22.043	56.863
	android.permission. USE_CREDENTIALS	18.937	49.041	android.permission.USE_CREDENTIALS	21.034	54.260
	android.permission.READ_PROFILE	13.567	35.134	android.permission.READ_PROFILE	15.074	38.884
	android.permission.GET_ACCOUNTS	10.039	25.999	android.permission.GET_ACCOUNTS	10.848	27.984
	android.permission. READ_EXTERNAL_STORAGE	6.582	17.044	android.permission.READ_EXTERNAL_ STORAGE	7.104	18.326
	android.permission.CAMERA	5.504	14.253	android.permission.CAMERA	5.831	15.043
	android.permission. ACCESS_NETWORK_STATE	3.895	10.087	android.permission.ACCESS_NETWORK_ STATE	4.005	10.331
	android.permission.SEND_SMS	3.393	8.786	android.permission.SEND_SMS	3.431	8.851
	android.permission.READ_SMS	2.376	6.152	android.permission.READ_SMS	2.435	6.280
	android.permission.ACCESS_ WIFI_STATE	2.017	5.224	android.permission.RECEIVE_SMS	1.672	4.312
Intent (Maximum 3 perturbations)	android.intent.action MY_PACKAGE_REPLACED	39.921	58.787	android.intent.action.MY_PACKAGE_ REPLACED	40.042	63.543
	android.intent.action.MEDIA_BUTTON	14.591	21.487	android.intent.action.MEDIA_BUTTON	14.982	23.775
	android.intent.action. TIMEZONE_CHANGED	8.182	12.049	android.intent.action.ACTION_POWER_ DISCONNECTED	7.878	12.502
	android.intent.action.SEND	6.881	10.133	android.intent.action.TIMEZONE_ CHANGED	7.721	12.252
	android.intent.action. PACKAGE_REPLACED	5.134	7.560	android.intent.category.BROWSABLE	6.641	10.538
	android.intent.action. ACTION_POWER_DISCONNECTED	5.094	7.501	android.intent.action.SEND	5.066	8.038
	android.intent.category.BROWSABLE	5.011	7.380	android.intent.category.MONKEY	2.756	4.374
	android.intent.category.DEFAULT	3.030	4.462	android.intent.action.PACKAGE_ REPLACED	2.656	4.215
	android.intent.action. PACKAGE_REMOVED	1.595	2.349	android.intent.action.SEARCH	2.368	3.758
	android.intent.category.MONKEY	1.519	2.236	android.intent.category.DEFAULT	2.347	3.725

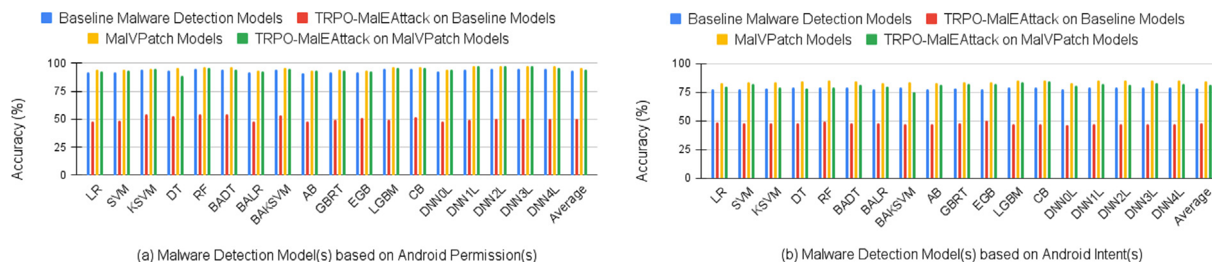


Fig. 5. Overall Performance of different Android Malware Detection Models (Permission/Intent) w.r.t. **TRPO-MalEAttack**

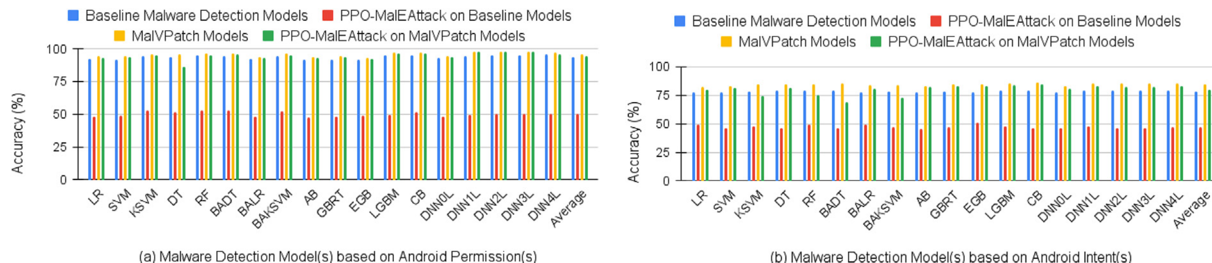


Fig. 6. Overall Performance of different Android Malware Detection Models (Permission/Intent) w.r.t. **PPO-MalEAttack**

models (93.55%) to MalVPatch models (95.57%). The highest accuracy among MalVPatch models (permission) is achieved by DNN1L (97.56%), whereas the lowest is attained by EGB (93.65%). Similarly, Fig. 5(b) shows the performance of MalVPatch malware detection models (intents) (yellow bar) on TRPO-MalEAttack models. The average accuracy of eighteen MalVPatch detection models (intent) improved by 5.90% from baseline models (78.47%) to MalVPatch models (84.37%). The highest accuracy among MalVPatch models (intent) is achieved by DNN2L (85.55%), whereas the lowest is attained by AB (83.20%).

On the other hand, Fig. 6(a) shows the performance of malware detection models (permission) after applying the **MalVPatch on**

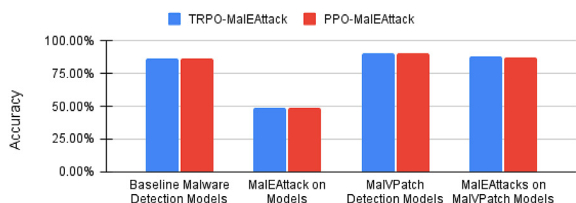


Fig. 7. Summary of *Average Accuracy* of baseline, MalEAttack, MalVPatch, MalEAttack on MalVPatch

PPO-MalEAttack models. We observe a similar accuracy improvement of MalVPatch malware detection models (yellow bar) over baseline models (blue bar). The average accuracy of eighteen MalVPatch models (permission) improved by 2.06% from baseline models (93.55%) to MalVPatch models (95.61%). The highest accuracy among MalVPatch models (permission) is achieved by DNN1L (97.67%), whereas the lowest is attained by EGB (93.14%). Similarly, Fig. 6(b) shows the performance of MalVPatch malware detection models (intents) (yellow bar) on PPO-MalEAttack models. The average accuracy of eighteen MalVPatch models (intent) improved by 5.96% from baseline models (78.47%) to MalVPatch models (84.43%). The highest accuracy among MalVPatch models (intent) is achieved by CB (85.76%).

4.3.2. Adversarial robustness @ MalVPatch models

We investigate the adversarial robustness of the MalVPatch malware detection models by performing a reattack using MalEAttacks on them. The reattack is performed in exactly the same way as the initial attack, with baseline models replaced by MalVPatch models.

Fig. 5 shows the performance of TRPO-MalEAgent reattack on MalVPatch malware detection models (green bar). The **TRPO-MalEAgent reattack** on eighteen MalVPatch detection models (permission) attain an average **fooling rate** of 1.82%, leading to an average **accuracy reduction** from 95.57% to 94.47%. Similarly, TRPO-MalEAgent reattack on eighteen MalVPatch models (intent) attain average fooling rate of 3.16% which leads to an average accuracy reduction from 84.37% to 81.05%.

On the other hand, Fig. 6 shows the performance of **PPO-MalEAgent reattack** on MalVPatch malware detection models (green bar). The PPO-MalEAgent reattack on eighteen MalVPatch models (permission) attain an average **fooling rate** of 1.97% leading to an average **accuracy reduction** from 95.61% to 94.40%. Similarly, PPO-MalEAgent reattack on eighteen MalVPatch models (intent) attain an average fooling rate of 5.57% which lead to an average accuracy reduction from 84.42% to 80.08%.

4.4. Summary

Fig. 7 shows the overall summary during the different stages of the proposed framework (Fig. 1). The thirty-six baseline android malware detection models achieved an average accuracy of 86.01%. The **TRPO-MalEAttack** exploited the vulnerabilities and reduced the average accuracy to 49.11% for the above detection models. The MalVPatch defense improves the average accuracy of the above models to 89.97%. Finally, the TRPO-MalEAgent Reattack on MalVPatch models achieved an average of 87.76% for thirty-six malware detection models.

Similarly, the **PPO-MalEAttack** reduced the average accuracy of thirty-six malware detection models from 86.01% to 48.65%. The MalVPatch defense improves the average accuracy of the above detection models to 90.02%. Finally, the PPO-MalEAgent Reattack on MalVPatch models achieved an average of 87.25% for thirty-six android malware detection models.

5. Related work

Existing literature suggests that ML/DL models in various domains have shown promising results but are vulnerable to adversarial attacks. Pitropakis et al. (2019); Machado et al. (2021); Deldjoo et al. (2021); Sewak et al. (2021) provide an extensive summary of the landscape of adversarial attacks and defenses in ML/DL models. However, unlike image classification, object detection, recommendation system etc., the adversarial robustness of android malware detection models has received minimal research

Table 3
Performance comparison of the proposed work with existing literature.

Author(s)	Attack Scenario	Max. No of Perturbations	No of Malware Detection Models	Fooling Rate (FR)	Vulnerability List	Defense Strategy	Adversarial Robustness
Grosse et al. (2016)	White box	20	Only DNNs	40.97%–84.05%	No	Feature Reduction Distillation	No
Hu and Tan (2017) (MalGAN)	White box	100%	6	99%	No	AT	No
Taheri et al. (2020) (Trivial)	White box	60	Total: 3 RF, BagDT, SVM	27.40%	No	AT, GAN	No
Taheri et al. (2020) (Distribution)	White box	60	Total: 3 RF, BagDT, SVM	27.98%	No	AT, GAN	No
Taheri et al. (2020) (KNN)	White box	60	Total: 3 RF, BagDT, SVM	27.43%	No	AT, GAN	No
Taheri et al. (2020) (ACO)	White box	60	Total: 3 RF, BagDT, SVM	27.08%	No	AT, GAN	No
Rathore et al. (2021a) (SPA)	White box	Permission: 5	8	Avg 8 models: 44.28%	No	AT	Yes
Proposed (TRPO-MalEAttack)	White box	Permission: 6 Intent: 3 Mean: 2.02	Total: 18 ML: 4, Bagging: 4 Boosting: 5, DNN: 5	Avg 36 models: 95.75%	Yes	MalVPatch	Yes
Proposed (PPO-MalEAttack)	White box	Permission: 6 Intent: 3 Mean: 2.06	Total: 18 ML: 4, Bagging: 4 Boosting: 5, DNN: 5	Avg 36 Models: 96.88%	Yes	MalVPatch	Yes

attention. Taheri et al. (2020) proposed five adversarial attacks and two defense strategies for the white-box scenario. However, they attained a very limited fooling rate of around 27% with 60 perturbations. They tested their attack and defense strategies with only three classification algorithms and did not discuss the adversarial robustness of the proposed defense. Rathore et al. (2021a) introduced the Q-learning-based single policy attack (SPA) for white box and multiple policy attack (MPA) for grey box scenario. However, they achieved average fooling rates of just 44.28% using SPA and 53.20% using MPA for eight classification algorithms. Grosse et al. (2016) crafted adversarial samples using neural networks. They achieved misclassification rates in the range of 40.97% to 84.05% with a maximum of 20 modifications. Also, the defense strategies proposed in the paper did not offer a significant reduction in the misclassification rates. Li et al. (2019) proposed the E-MalGAN black box attack based on bi-objective GAN, which is an improvement over the MalGAN attack proposed by Hu and Tan (2017).

Table 3 compares the proposed work with the existing literature. The adversarial attack aims to add minimum perturbations in malware samples while maximizing the fooling rate against various malware detection models. Most of the adversarial attacks in the literature add massive perturbations to generate adversarial samples, which increase the attack's cost. On the other hand, they achieved minimal success in generating adversarial samples and thus have limited fooling rates. The adversarial attack and defense strategies should generalize well in the ecosystem. Authors have tested their attacks and defense strategies on a very limited number of classification algorithms, feature sets, etc. Some attack/defense strategies are only restricted on DNNs, which is a huge drawback. On the other hand, authors have rarely shared the perturbation details that add explainability to work. The ultimate aim of these studies is to develop adversarial superior malware detection models. However, limited authors have calculated the adversarial robustness in malware detection.

6. Conclusion and future work

This work aims to develop adversarially superior android malware detection models using a five-step proactive adversary-aware framework. We first constructed thirty-six different malware detection models with an average accuracy of 86.01%. Then we designed two targeted false-negative evasion attacks based on RL against the above detection models. The TRPO-MalEAttack against thirty-six malware detection models achieved an average fooling rate of 95.75% (with 2.02 mean perturbations) and reduced their average accuracy from 86.01% to 49.11%. Similarly, The PPO-MalEAttack against thirty-six malware detection models achieved a higher average fooling rate of 96.87% (with 2.08 mean perturbations) and reduced their average accuracy from 86.01% to 48.65%. We also compiled a list of *TEN* android permissions and intents that are most vulnerable and can be used by adversaries to generate more adversarial applications. The proposed MalVPatch defense improves the detection accuracy and drastically enhances the adversarial robustness of malware detection models. Finally, we conclude that investigating the adversarial robustness of ML/DL based malware detection models is an essential step before their real-world deployment. It also helps in achieving adversarial superiority in malware detection.

In the future, we plan to investigate collusion based adversarial attacks against malware detection models. We also plan to study the adversarial robustness of other features along with clustering and hybrid malware detection models. We plan to explore a game-theoretical approach to stay adversarially superior in malware detection.

References

Arora, A., Peddoju, S.K., Conti, M., 2019. Permpair: android malware detection using permission pairs. IEEE Trans. Inf. Forensics Secur. 15, 1968–1982.
Deldjoo, Y., Noia, T.D., Merra, F.A., 2021. A survey on adversarial recommender systems: from attack/defense strategies to generative adversarial networks. ACM Comput. Surv. 54, 1–38.
Global Digital Report, 2022. Simon kemp (hootsuite). Available: https://www.hootsuite.com/resources/digital-trends. (Accessed January 2021).
Goodfellow, I.J., Shlens, J., Szegedy, C., 2015. Explaining and harnessing adversarial examples. In: International Conference on Learning Representations (ICLR).
Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P., 2016. Adversarial Perturbations against Deep Neural Networks for Malware Classification arXiv preprint arXiv:1606.04435.
Hu, W., Tan, Y., 2017. Generating Adversarial Malware Examples for Black-Box Attacks Based on gan arXiv preprint:1702.05983.
Khariwal, K., Singh, J., Arora, A., 2020. Ipdroid: android malware detection using intents and permissions. In: Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4). IEEE, pp. 197–202.
Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., Ye, H., 2018. Significant permission identification for machine-learning-based android malware detection. IEEE Trans. Ind. Inf. 14, 3216–3225.
Li, H., Zhou, S., Yuan, W., Li, J., Leung, H., 2019. Adversarial-example attacks toward android malware detection system. IEEE Syst. J. 14, 653–656.
Machado, G.R., Silva, E., Goldschmidt, R.R., 2021. Adversarial machine learning in image classification: a survey toward the defender's perspective. ACM Comput. Surv. 55, 1–38.
McAfee Mobile Threat Report, 2022. Available: https://www.mcafee.com/content/dam/consumer/en-us/docs/reports/rp-mobile-threat-report-feb-2022.pdf. (Accessed April 2022).
Pitropakis, N., Panaousis, E., Giannetsos, T., Anastasiadis, E., Loukas, G., 2019. A taxonomy and survey of attacks against machine learning. Computer Science Review (CSR) 34, 100199.
Rathore, H., Sahay, S.K., Rajvanshi, R., Sewak, M., 2020. Identification of significant permissions for efficient android malware detection. In: EAI BROADNETS. Springer, pp. 33–52.
Rathore, H., Sahay, S.K., Nikam, P., Sewak, M., 2021a. Robust android malware detection system against adversarial attacks using q-learning. Inf. Syst. Front 1–16.
Rathore, H., Sahay, S.K., Thukral, S., Sewak, M., 2021b. Detection of malicious android applications: classical machine learning vs. deep neural network integrated with clustering. In: EAI BROADNETS. Springer, pp. 109–128.
Rathore, H., Samavedhi, A., Sahay, S.K., Sewak, M., 2021c. Robust malware detection models: learning from adversarial attacks and defenses. Forensic Sci. Int.: Digit. Invest. 37, 301183.
Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P., 2015. Trust region policy optimization. In: International Conference on Machine Learning (ICML), PMLR., pp. 1889–1897.
Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal Policy Optimization Algorithms arXiv preprint arXiv:1707.06347.
Sewak, M., Sahay, S.K., Rathore, H., 2020. Deepintent: implicitintent based android ids with e2e deep learning architecture. In: IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), IEEE, pp. 1–6.
Sewak, M., Sahay, S.K., Rathore, H., 2021. DRLDO: a novel DRL based de-obfuscation system for defence against metamorphic malware. Defence Sci. J. 71, 55–65.
Sophos Threat Report, 2022. Available: https://assets.sophos.com/X24WTUEQ/at/b739xqx5jg5w9w7p2bpzxg/sophos-2022-threat-report.pdf. (Accessed April 2022).
Taheri, R., Javidan, R., Shojafar, M., Vinod, P., Conti, M., 2020. Can machine learning model with static features be fooled: an adversarial machine learning. Cluster Comput. 1–21.
Test, A.V.-, 2022. Total amount of malware and pua under android. Available: https://portal.av-atlas.org/malware/statistics. (Accessed April 2022).
Zhu, H., Li, Y., Li, R., Li, J., You, Z., Song, H., 2020. Sedmdroid: an enhanced stacking ensemble framework for android malware detection. IEEE.Trans.Netw. Sci.Eng 8, 984–994.
Zimperium Global Mobile Threat Report, 2022. Available: https://www.zimperium.com/global-mobile-threat-report/. (Accessed April 2022).
Zou, D., Wu, Y., Yang, S., Chauhan, A., Yang, W., Zhong, J., Dou, S., Jin, H., 2021. Intdroid: android malware detection based on api intimacy analysis. ACM Trans. Software Eng. Methodol. 30, 1–32.