



# Detection and robustness evaluation of android malware classifiers

M. L. Anupama<sup>1</sup> · P. Vinod<sup>2</sup> · Corrado Aaron Visaggio<sup>3</sup> · M. A. Arya<sup>1</sup> · Josna Philomina<sup>1</sup> · Rincy Raphael<sup>4</sup> · Anson Pinhero<sup>1</sup> · K. S. Ajith<sup>1</sup> · P. Mathiyalagan<sup>4</sup>

Received: 7 August 2020 / Accepted: 31 May 2021 / Published online: 26 June 2021  
© The Author(s), under exclusive licence to Springer-Verlag France SAS, part of Springer Nature 2021

## Abstract

Android malware attacks are tremendously increasing, and evasion techniques become more and more effective. For this reason, it is necessary to continuously improve the detection performances. With this paper, we wish to pursue this purpose with two contributions. On one hand, we aim at evaluating how improving machine learning-based malware detectors, and on the other hand, we investigate to which extent adversarial attacks can deteriorate the performances of the classifiers. Analysis of malware samples is performed using static and dynamic analysis. This paper proposes a framework for integrating both static and dynamic features trained on machine learning methods and deep neural network. On employing machine learning algorithms, we obtain an accuracy of 97.59% with static features using SVM, and 95.64% is reached with dynamic features using Random forest. Additionally, a 100% accuracy was obtained with CART and SVM using hybrid attributes (on combining relevant static and dynamic features). Further, using deep neural network models, experimental results showed an accuracy of 99.28% using static features, 94.61% using dynamic attributes, and 99.59% by combining both static and dynamic features (also known as multi-modal attributes). Besides, we evaluated the robustness of classifiers against evasion and poisoning attack. In particular comprehensive analysis was performed using permission, APIs, app components and system calls (especially *n*-grams of system calls). We noticed that the performances of the classifiers significantly dropped while simulating evasion attack using static features, and in some cases 100% of adversarial examples were wrongly labelled by the classification models. Additionally, we show that models trained using dynamic features are also vulnerable to attack, however they exhibit more resilience than a classifier built on static features.

**Keywords** Static features · Dynamic features · Hybrid features · Fisher score · Adversarial examples · Attack models

## 1 Introduction

Malicious code is a software intentionally written for bypassing security controls and performing unauthorized actions that are not allowed to the attacker and can cause a damage to the victim. The techniques for analyzing malicious code can be divided into static analysis and dynamic analysis. Static analysis techniques scan the source code and don't require

---

✉ Corrado Aaron Visaggio  
visaggio@unisannio.it

M. L. Anupama  
anupama.ml@scmsgroup.org

P. Vinod  
vinod.p@cusat.ac.in

M. A. Arya  
aryanand54@gmail.com

Josna Philomina  
josnaphilomina@scmsgroup.org

Rincy Raphael  
rincyraphael2019@sec.ac.in

K. S. Ajith  
ajithks273@gmail.com

P. Mathiyalagan  
mathiyalagan.p@sec.ac.in

<sup>1</sup> Present Address: Department of Computer Science and Engineering, SCMS School of Engineering and Technology, Cochin, India

<sup>2</sup> Department of Computer Applications, Cochin University of Science and Technology, Cochin, India

<sup>3</sup> Department of Engineering, University of Sannio, Benevento, Italy

<sup>4</sup> Department of Computer Science and Engineering, Sri Ramakrishna Engineering College, Coimbatore, Affiliated by Anna University, Chennai, India

the execution of the programs to be examined. Thus, the study can be conducted without compromising the systems. Static analysis gained wider acceptance amongst the analysts as it is quick and harmless, even though encryption, obfuscation and the use of runtime libraries obstruct the static analysis. Dynamic analysis, on the contrary, aims to uncover the runtime behaviour of the application by executing the application on the real device or in a sandbox environment [8]. Dynamic analysis is not limited by code obfuscation and can provide details about the malware behavior.

By combining both static and dynamic analysis, it is possible to leverage the advantages of both approaches: malware scanners that use both the types of analysis are generally known as hybrid malware detectors. Static analysis is conducted by extracting structural features from the file, while dynamic analysis uses features that require the execution of the app, like system calls, network traces, and control flow graphs.

Despite the large literature investigating the advantages and limitations of using machine learning for detecting malware, further studies are necessary for consolidating the body of knowledge on this topic and removing all the uncertainties research pointed out so far, for different reasons. Recent works collect evidence that anti-malware tools are diminishing their ability to recognize malware, due mainly to the rapid increment of variants [20,40,50]. Spatial and temporal bias can make untrustworthy some results, since training or testing sets are not completely representative of the malware (and goodware) population [34]. Adversarial attacks could easily deteriorate the robustness of machine and deep learning based classifiers [10,19], while there is not a complete convergence about which are the best machine learning algorithms for malware detection [16,44]. For this reason with this paper we aim at providing a two-fold contribution to the state of the art: adding further evidence about the performance of machine and deep learning algorithms in detecting malware, and studying to which extent adversarial samples may alter the effectiveness of classifiers.

More in detail, in this work we explore the usage of the Fisher score [52] to select the most relevant attributes for the classifiers. The features obtained are used to build diverse classifiers using Logistic Regression, Classification and Regression Trees, Random Forest and Support Vector Machine algorithms. A comprehensive analysis of the machine learning models is conducted to identify the optimal classification model that can be deployed for detecting unseen or future samples. Finally, we realized three attack models which leverage adversarial examples and evaluate how the classifiers performances degrade. We observed that a minor perturbation of attributes significantly dropped the detection rate, and all the modified malware samples (tainted/adversarial examples) bypassed the detection.

Finally, the main contributions of this research work are as follows:

- We implement a feature selection algorithm based on Fisher score for ranking attributes, and show that classifiers trained on the relevant attributes selected in this way can improve the detection rate.
- We create multi-modal features (hybrid features) classifiers and obtain an accuracy of 100% with CART, SVM, and an accuracy of 99.59% with deep neural network.
- We realize three attack models based on hamming distance, k-means and app's components for creating adversarial samples. These specimens are created by inserting permissions and app's components into malicious apps. We observed that classifiers' performance dropped drastically. In particular, Hamming distance based attack increases the average False Positive Rate of machine learning classifiers and deep neural network by 55.86% and 45.94% respectively. All the adversarial samples developed using k-means clustering are successfully evaded ( $FNR = 100\%$ ). Finally, 90.13% and 100% tainted applications created by injecting especially crafted app's components deceived classifiers based on machine learning approaches and deep neural network.

The paper is organized as follows. Section 2 discusses the related work. In Sect. 3, proposed methodology is presented. The adversarial attacks are introduced in Sect. 4 while the attacks are elaborated in Sect. 5. The experiments and obtained results are given in Sect. 6. Evaluation on obfuscated samples are discussed in Sect. 7. Finally, the concluding remarks and direction for future work is given in Sect. 8.

## 2 Related work

This section discusses existing malware detection and classification models based on both machine learning and deep learning. Patel *et al.* [33] proposed a hybrid android malware detection system. It extracts both permission and behaviour-based features. Then, performed feature selection using information gain. Finally, rule generation module classifies applications as benign or malicious. In [46], authors have mentioned another hybrid malware detector that uses SVM classifier to classify app as benign or malware. It detects zero-day malware with a true positive rate of 98.76%. Damodaran *et al.* [16] conducted a comparative analysis on malware detection system employing static, dynamic, and hybrid analysis. They found that behavioural data produce an highest AUC of 0.98 using Hidden Markov Models (HMMs) trained on 785 samples. In [47], authors initially utilize APIMonitor to obtain static features from apps. Then, it involves the usage of APE\_BOX to obtain dynamic features. Finally, they

apply SVM for classification. MADAM [38] demonstrated how KNN classifier can achieve 96.9% detection rate.

Significant Permission Identification, SigPID [27] is another malware detection system that uses a three-layered pruning by mining the permission data to identify the most significant permissions that result in differentiating benign and malicious apps. It then uses machine-learning classifiers (SVM and decision tree) for classification and achieved over 90% of detection accuracy. In [15], authors initially disassemble applications by using Androguard to obtain the frequency of API calls used by the application. Finally, it is observed that a particular set of APIs is more frequent in malicious apps. It can detect malicious apps with 96.69% accuracy and 95.25% detection rate, by using SVM.

Crowdroid [11] is an Android malware detector which uses dynamic analysis and then employs two-means clustering algorithm for classifying benign and malicious apps. In [18], authors have presented an Android Malware Detection system which extracts system calls by executing the applications in a sandbox environment. They implemented their approach in MALINE tool and can detect malware with low rates of false positives by employing machine learning algorithms. Afonso *et al.* [2] propose another android malware detection system that uses dynamic features such as API calls and system call traces along with machine learning to identify malware with high detection rate.

Authors in [21] present a machine-learning-based Android malware detection and family identification approach, RevealDroid, that aims at reducing the sets of features used in the classifiers. This approach leverages categorized Android API usage, reflection-based features, and features from native binaries of apps. Besides accuracy and efficiency, authors evaluate also obfuscation resilience using a large dataset of more than 54,000 malicious and benign apps. The experimental results show an accuracy of 98.

Tam *et al.* [43] propose a mechanism for reconstructing behaviors of Android malware by observing and dissecting system calls. This mechanism allows CopperDroid to obtain events of interest, especially intra- and inter-process communications. This makes CopperDroid agnostic to the underlying invocation methods. Experimental results showed that CopperDroid discloses additional behaviors on more than 60% of the analyzed dataset.

In [4] authors analyze the permissions used by an application that requires during installation. It uses clustering and classification techniques and also allows user to identify malicious applications installed on the phone and also provides a provision to remove them. The drawback of this system is that if a new unknown family of a malware is supposed to be detected then a new cluster has to be created considering the same family's permission. CSCdroid [51] builds a Markov chain by using system calls. Then, it constructs the target feature vector from the probability matrix.

Finally, it uses the Support Vector Machine classifier to detect malware, achieving an F1-score of 98.11% and a true positive rate of 97.23%.

Kimet *al.* [25] propose an Android malware detection method, that uses opcode features, API features, strings, permissions, app's components, and environmental features, to generate a multimodal malware detection model. With these static features, they trained their initial networks. Later, they trained the final network, with initial network results. The model produces an accuracy of 98%. Paper [41] proposes a malware detection model-based on RNN and CNN. It involves the usage of the static feature opcode. Finally authors conclude that their accuracy exceeds 92%, for even small training datasets. Malware Classification using SimHash and CNN, MCSC [32] is a model leveraging opcode sequences as static features, that combine malware visualization, and deep learning techniques, resulting in a classification accuracy of 99.26%.

In [39] authors propose a deep neural network-based malware detector using static features. It consists of three components, the first component extracts features, the second component is a DNN classifier, and the final component is a score calibrator which translates the output of a NN to a score. Achieved 95% detection rate, at 0.1% false-positive rate (FPR). MalDozer [24] is another highly accurate malware detection model that relies on deep learning techniques and raw sequences of API method calls. Deep android malware detection [29] is another model developed based on the static analysis of the raw opcode sequence from a disassembled program. Features indicative of malware are automatically learned by the network from the raw opcode sequence thus removing the need for hand-engineered malware features. This model has proposed a much simpler training pipeline.

A comprehensive analysis and comparison of deep neural networks(DNNs) and various classical machine learning algorithms for static malware detection are discussed in [45]. The authors have concluded that DNNs perform comparably well and are well suited to address the problem of malware detection using static PE features. A malware classification method using Visualization and deep learning is mentioned in [26]. It requires no expert domain knowledge. Initially, the files are visualized as grayscale images then experimented on deep learning architectures involving different combinations of Convolutional Neural Network (CNN) and Long Short Term Memory (LSTM). A deep learning approach that amends the convolutional deep learning models to use the support vector machine is presented in [3]. The authors have finally concluded that, among their three models, the model with 5 layers has the best accuracy compared to those with 2 and 3 layers.

A CNN based windows malware detector that uses API calls and their corresponding category as dynamic features that finally resulted in the achievement of 97.97% accuracy

for the N-grams counselled by the Relief Feature Selection Technique is described in [36]. In [28], the authors have designed a method based on a convolutional neural network applied to the system calls occurrences through dynamic analysis. They obtained an accuracy ranging between 0.85 and 0.95.

In [48], authors have presented a method based on back-propagation neural network to detect malware. It builds Markov chains from system call sequences and then applies the back-propagation neural network to detect malware. They experimented on a dataset of 1,189 benign ones, and 1,227 malicious applications and obtained an F1-score of 0.983.

KuafuDet [14] is a two-phase detection system, where features are extracted in the first phase and the second phase is an online detection phase. Camouflaged malicious applications which is a form of adversarial examples are developed, and similarity-based filtering is used to identify false negatives.

Xu et al. [49] applied genetic programming for evading PDF malware classifiers. It uses the probabilities assigned by the classifiers to estimate the fitness of variants. PDFrate and Hidost were the two PDF classifiers used for the evaluation. Authors reported 500 evasive variants created in 6 days. The evaluation of adversarial attacks were performed on Android malware detectors. Authors in [13] proposed a system called DroidEye which extracts features from Android apps and represents each observation as a binary vector. Further, they evaluated the attack on standard classifiers used for identifying malware. To improve the robustness of these classifiers, they transformed the binary vectors as continuous probabilities. Experiments were performed on samples collected from Comodo Cloud Security Center, and reported that DroidEye improved the security of system without effecting the detection performance. Adversarial crafting attacks on neural network were experimented in [23]. The attack was demonstrated on malware detection system trained on DREBIN dataset, where each application was represented as a binary vector. They reported a classification accuracy of 97% with FNR value of 7.6%. The trained model was subjected with adversarial examples generated by modifying AndroidManifest.xml and achieved a misclassification rate of 85%. In addition, they hardened neural network using adversarial training and defensive distillation, and reported that the later approach reduced the misclassification rates. Comprehensive experiments considering permissions [12] were performed for binary classification (malware vs benign) and multi class classification. Their study demonstrated that carefully selecting permissions can lead to accurate detection and classification. Further, to evaluate robustness of permission-based detection, top benign permissions were added to the malicious applications. They showed that a small number of requested benign permissions decreases ANN performance. However, ANN recovers on larger permission

request, indicating identical performance as observed with unmodified malware applications.

Demontis et al. [17] developed an adversary-aware machine learning detector against evasion attacks. Authors propose a secure learning solution which is able to retain computational efficiency and scalability on large datasets. The method outperforms state of the art classification algorithms, without loss of accuracy when there aren't well-crafted attacks.

Pierazzi et al. [35] propose a formalization of problem-space attacks. They uncover new relationships between feature space and problem space, providing necessary and sufficient conditions for the existence of problem-space attacks. This work shows that adversarial malware can be produced automatically.

In our work, we build machine learning and deep learning models using static, dynamic and hybrid techniques. We found that DNN obtained better performance using hybrid features. Further, we conducted comprehensive analysis on adversarial attacks by proposing three approaches for creating adversarial examples, and conclude that malware classifiers can be easily defeated by introducing tiny perturbations.

The Table 13 summarizes the main contributions of each analyzed work.

## 3 Methodology

This section describes the methods used by the hybrid malware detector that we will study.

### 3.1 Static analysis

An Android application explicitly requires the user to approve the necessary permissions during the installation. As a consequence, the collection of permissions can reflect the application behaviour. Standard and non-standard permissions may be extracted from AndroidManifest.xml file using Android Asset Packaging Tool (aapt) command. We developed a parser to read the manifest file to extract `<user-permission>` and the `<permission>` tag containing the permission name. Besides, our parser captures the application components: activities, services, broadcast receivers, and content providers are obtained by decompiling the given app using APKTool [6].

### 3.2 Dynamic analysis

We use dynamic analysis for capturing the sequence of system calls while the application executes and interacts with the operating system. Given an Android application, the procedure for extracting system calls is shown in Fig. 1. Initially the application is installed in Nexus\_5\_API\_22 Android emula-

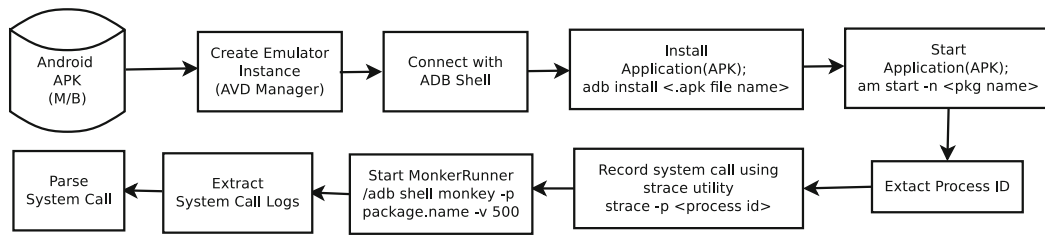


Fig. 1 System call extraction

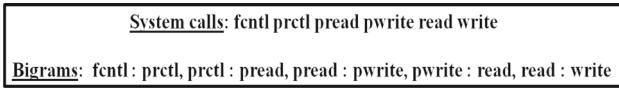


Fig. 2 Sample set of system calls and bigrams

tor using the ADB install command. The Monkey tool [30] is used for interacting with the application and generating system calls. Monkey tool is configured to give automatic user inputs (events) which are: making a call, sending SMS, changing geo location, updating the battery charging status, incoming call 200 times in a minute. Then system calls are recorded using the strace utility. Once the specified events are completed, the application is uninstalled with the ADB uninstall command, and the emulator is set into a clean state for the next app installation. We consider system call names and ignored the parameters of call. In order to avoid the presence of rare system calls in the feature space, we collected five execution traces for each applications. We noticed a longer call trace in case of benign application compared to malicious apps. Also, we noticed top 10 frequently invoked system calls in malicious applications were brk, bind, fchown32, sendto, gettimeofday, epoll\_wait, getuid, getpid, clock\_gettime and mprotect.

### 3.3 System call bigram generation

Bigrams are generated from the obtained system calls in a separate text document for each application. Figure 2 shows a sample set of features (system calls) and their corresponding bigrams. The detailed architecture of the proposed Hybrid Malware Detector is shown in Fig. 3.

### 3.4 Fisher score algorithm

In order to select the most relevant features, an algorithm was developed implementing the Fisher-score.

The algorithm takes as input a set of system calls. Initially, the mean for benign samples is computed, then for malware samples; the variance for benign and malware samples is obtained. The Fisher score is computed for benign and for malware samples. Finally, the Fisher scores obtained are

### Algorithm 1 Fisher score algorithm

**Input:**  $F = \{f_1, f_2, f_3 \dots f_m\}$  where  $f_i$  represents a feature

**Output:**  $D = \{f_1, f_2, f_3 \dots f_k\}$  where  $k \ll m$

```

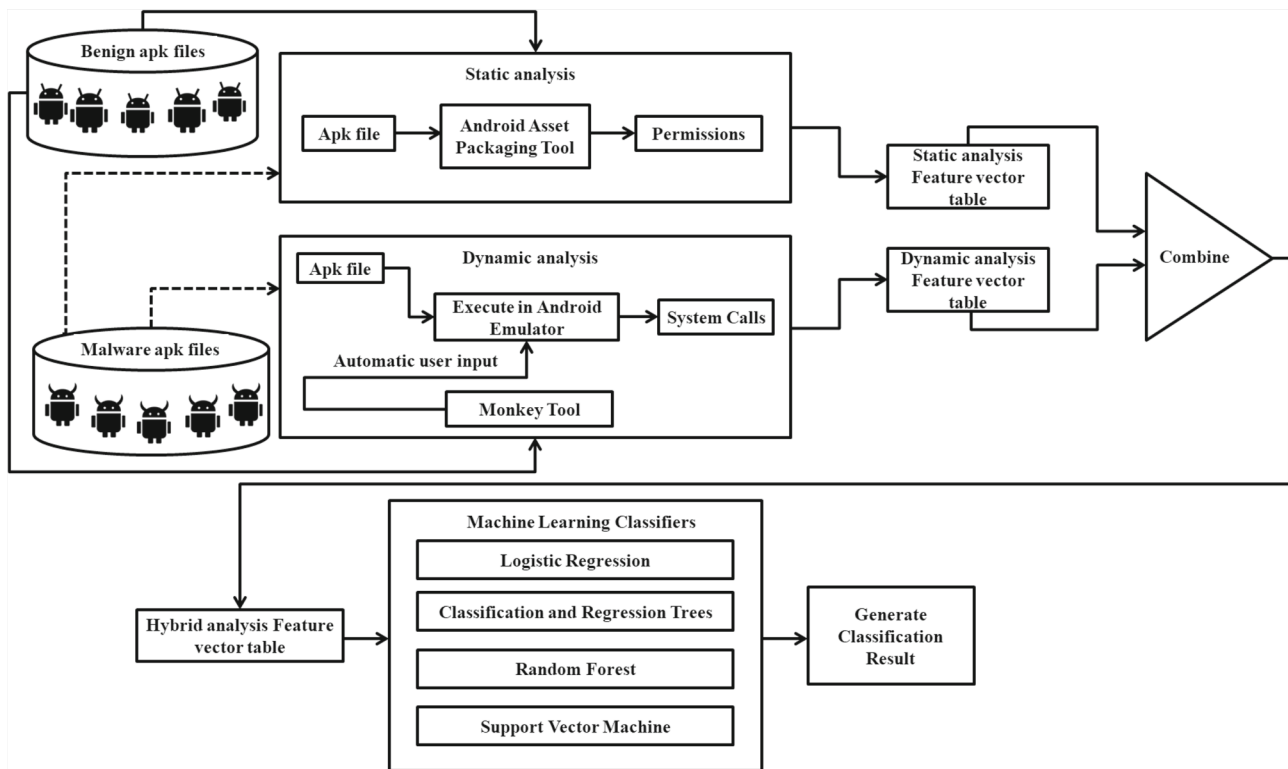
1: Start
2: for i = 1 to m do
3:    $\mu_B = n_m(\mu_{f_i}^B - \mu_{f_i})^2$                                 > Mean
4: end for
5: for i = 1 to m do
6:    $\mu_M = n_m(\mu_{f_i}^M - \mu_{f_i})^2$                                 > Mean
7: end for
8: for i = 1 to m do
9:   for j = 1 to n do
10:     $\sigma_B = (f_{ji} - \mu_{f_i}^B)^2$                                 > Variance
11:   end for
12: end for
13: for i = 1 to m do
14:   for j = 1 to n do
15:     $\sigma_M = (f_{ji} - \mu_{f_i}^M)^2$                                 > Variance
16:   end for
17: end for
18:  $F(f_i)_b = \frac{\mu_B}{\sigma_B}$                                     > Fisher score
19:  $F(f_i)_m = \frac{\mu_M}{\sigma_M}$                                     > Fisher score
20:  $F(f_i)_{bm} = \frac{\mu_B + \mu_M}{\sigma_B + \sigma_M}$                 > Fisher score
21: Sort the fisher scores obtained in descending order.
22: Stop
    
```

sorted in descending order. The steps involved are shown in Algorithm 1.

### 3.5 Features vector table generation

A sample features vector table is a dataframe consisting of a collection of features.  $F_1, F_2, F_3, \dots, F_p$ , represent 'p' features (permissions, system calls or app's component).  $S_1, S_2, S_3, \dots, S_q$  represent 'q' samples. Class labels in the last column are represented as either '0' or '1'. '0' denotes a benign app while '1' denotes a malware. The values in the table denoted by  $v_{11}, v_{12}, \dots, v_{qp}$  refer to the occurrence of a particular feature in a sample. In the case of static features, the occurrence of an attribute is represented by '1' while the absence of an attribute is represented by '0'. While in case of dynamic features and app's components, the elements of vectors are the number of times the  $p^{th}$  system call or the app's component was invoked by the  $q^{th}$  sample.

In the case of hybrid analysis, the features vector tables produced by both static and dynamic analysis are combined.



**Fig. 3** The architecture of the proposed Hybrid Malware Detector

$F_1, F_2, F_3, \dots, F_p$  represent the relevant attributes obtained after the features selection phase.

### 3.6 Machine learning unit

The training set is given to the classifier in the form of features vector table. Test data are supplied to it, thus the trained model assigns class labels to each sample in the test set. Here machine learning is run on features obtained with the static analysis (considering the permissions as the features), dynamic analysis (in this case the features are the system call bigrams), and hybrid analysis (permissions and system call bigram are used jointly). Each of these features vector tables is given as input to machine learning classifier and the performances of the different machine learning classifiers are compared. Also the features vector table generated after features selection based on Fisher score is given as input to machine learning and the obtained performances are compared.

#### 3.6.1 Training and testing

There are different techniques for training and testing. One is train-test split and the other is cross validation. In train test split, data are loaded in, then are split into training and test sets. The model is finally fitted to the training data. The

predictions are based on the input training data while are tested on the test data. With cross validation the dataset is split into  $k$  subsets:  $k - 1$  of these subsets are used for the training while the last subset is hold for test. For our experiment  $k$  is fixed to 10.

#### 3.6.2 Classifiers

Classification is a supervised learning approach, i.e. each sample of the training set is explicitly assigned to a category identified by a label. A classifier is an assumption or a function with discrete values that is used to assign class labels to input test samples. The machine learning classifiers in the proposed system used: the Logistic Regression (LR), Classification and Regression Trees (CART), Random Forest (RF), and Support Vector Machine (SVM). The features vector table is the input to the machine learning unit, which then generates the trained model, used to assign class labels to the samples of the test dataset.

## 4 Adversarial attacks on classifier

In the previous experiments, we discussed feature engineering for developing a classification model to accurately detect malware and benign apps. In this section, we discuss how

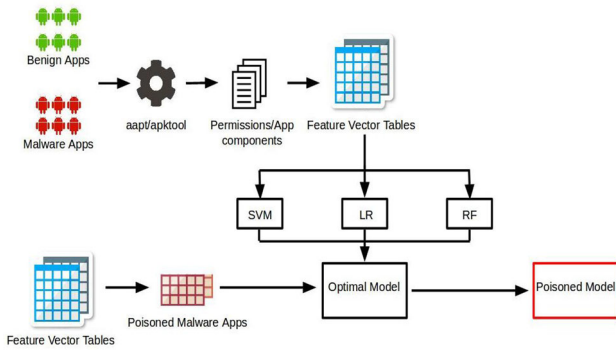


Fig. 4 Adversarial attack

adversarial attacks degrade the robustness of machine learning classifiers, thus we proposed three attack models. In the first phase, we develop the models for classification. Next we perform a poisoning attack on the optimal model. Figure 4 depicts the architecture of the proposed method. The dataset consists of benign applications and malware. Features such as permissions and app components are extracted using `aapt` and `apktool`. Using the extracted features, Feature Vector Tables (FVT) are created. The FVTs are given as input to the machine learning classifiers and DNN for training. In the next phase, the attack is launched on the classifiers. For the attack, 10% of total malware apps are chosen randomly as the test set. Hamming distance and KMeans clustering techniques are used for injecting additional permissions to malicious seed samples. App components are inserted by adding a perturbation in the FVT of app components. Attacks are explained in Sect. 5. The adversarial malware samples are presented to the trained model for predicting the modified applications. Further, we compute the performance of DNN when supplying adversarial samples. The classification accuracy, F1-score, precision and recall of the classifiers are evaluated before and after the attacks. We found that the classification accuracy of the classification model dropped to 40% and 10% for permissions and app components respectively.

### 4.1 Feature extraction

After data collection, features extraction is performed. In this approach, static features such as permissions and app components are extracted.

For extracting permissions, Android Asset Packaging Tool (AAPT) utility is used, which helps us to view, create and update zipped packages. To extract app’s components, applications are disassembled using `apktool`. `Apktool` is an utility for reverse engineering Android applications resources (APK).

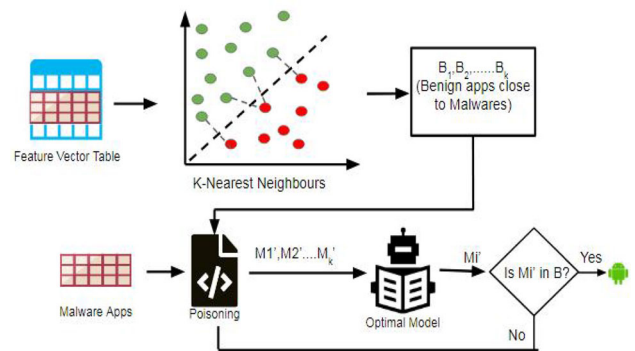


Fig. 5 Evasion Attack based on Hamming Distance

## 5 Evasion attack

Evasion attack is the process of injecting certain perturbations at test time to increase the error rate of the machine learning classifiers. Initially, classifiers say  $H$  is trained using dataset  $D = (X_i, y_i)_{i=1}^n$ , where  $X_i \in [1, 0]^d$  is a  $d$  dimensional feature vector for permissions and  $X_i \in [integer]^4$  is a four-dimensional feature vector since there are four app components.  $y_i \in [1, 0]$  are the class labels where  $i \in [1, \dots, n]$ . When the dataset is given to the classifiers as input, it performs a classification and response  $y$  is generated by  $s.t.H(X) = y$ . The goal of the is to add a small perturbation to feature vectors of  $X$ ,  $H(X + \mu) = H(X^*)$  such that  $H(X^*) = y'$  and  $y' \neq y$ . For permissions the perturbation  $\mu \in [1, 0]$  and for app components, the perturbation is  $X_{ij\_avg}$  or  $X_{ij\_max}$ , where  $X_{ij\_avg}$  is the average of an app component values in the dataset  $D$  and  $X_{ij\_max}$  is the maximum of an app component values in the dataset  $D$ .

Three types of attacks are proposed in this study using (a) hamming distance (b) K-means and (c) statistical methods. In the attack scenario, an adversary will add extra attributes to each malicious samples in the test set, until the classifiers wrongly labels suspicious files as legitimate. For the interest of deceiving classifiers, discriminant attributes characteristic of legitimates apps are inserted in the malware applications. In this context by discriminant attributes, we refer to subset of prominent features in one class but at the same time this set is rarely used in alternate class or vice-versa. This will result the decision boundaries of the target classes to overlap thereby increase misclassification.

### 5.1 Attack using hamming distance

The Hamming distance-based attack is performed using permissions. The attack model is shown in Fig. 5. A set of malware sample is randomly chosen as a test set. In the next step, the Hamming distance between a malwares in the test set and all benign samples are calculated.

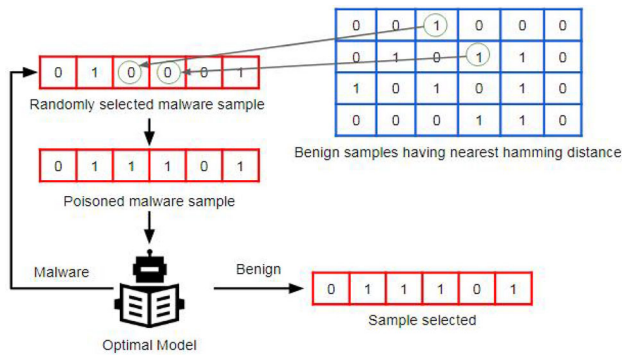


Fig. 6 Evasion attack an example

For example, let the feature vector of malware sample be  $M = 1011011001$  and that of benign be  $B = 0100110011$ . The Hamming distance between  $M$  and  $B$  is  $d(1011011001, 0100110011)$ , i.e.

$$1011011001 \oplus 0100110011 = 111111010$$

$$d(1011011001, 0100110011) = 7$$

The benign samples are arranged in ascending order of the distance with the malware seed sample. 0.5% of legitimate files that are close to the malwares are selected. Finally, the attack is performed on selected malware having feature vectors nearly identical to the legitimate app vectors. As the comparison performed over the entire feature space is computationally expensive. Hence, we randomly choose features, and if an attribute is present in benign (logic 1) and absent in malware (logic 0), then that feature is added to the malware sample. Figure 6 shows the addition of permissions to a malware sample.

Steps for adding features to the malware sample are:

- Select a malware sample from the test set.
- 0.5% of the nearest benign samples are shortlisted after calculating the Hamming distance.
- Perform XOR operation between the malware sample and the first benign sample in the shortlist.
- Randomly select an index where XOR gives a logic 1 as output.
- If the selected index has a logic 1 in a benign sample and logic 0 in the malware sample, then add a 1 to the corresponding index in the malware sample to get a new sample.
- The new sample is given to the optimal model for classification.
- If all of the three classifiers in the model predict the new sample as a benign one, then malware is selected and continue the iteration. Otherwise, randomly choose an alternate index, and compare its value in both malware and benign samples.

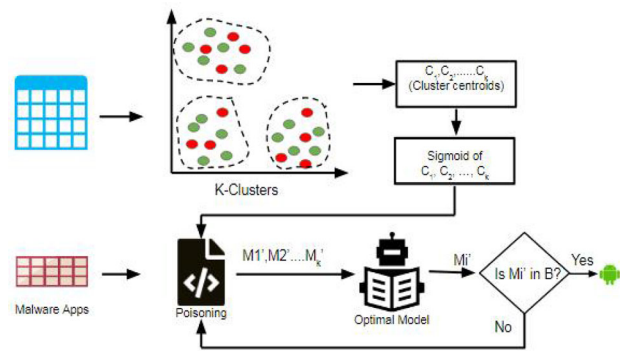


Fig. 7 Poisoning Attack Using KMeans Clustering

- These modified samples are presented to DNN for prediction, finally, the performance of DNN is recorded.

In the algorithm 2, lines 5 to 13 show step for calculating Hamming distance, which are stored in a two-dimensional array  $A$  of  $n$  rows and 2 columns, where  $n$  equals the number of benign samples. The elements of the first column indicate a benign vector and the second column is the Hamming distance to the malware sample. In line 14, values are sorted in ascending order to obtain the legitimate files close to the malware sample. The XOR operation in line 20 is computed to restrict unnecessary comparisons in future. The aim is to obtain the index of a feature that is present in a benign but absent in malware samples.

## 5.2 Evasion attack using KMeans clustering

In this approach, we cluster benign applications using K-Means clustering. The groups or clusters are formed by representing each legitimate application as a vector of permissions. The attack model is presented in Figure 7 and steps involved are described in algorithm 3. Further, the process of creating adversarial examples using K-Means is discussed below:

1. Randomly choose  $k$  centroids.
2. Calculate the Euclidean distance of malware seed sample to the centroids.
3. Assign each seed to the closest centroid and update the centroids by finding the mean value of all the data points in the cluster. This way we cluster all seed examples to the clusters which have similarity based on explicit permission declaration.
4. Compute XOR operation of each seed sample with the centroid vector.
5. Randomly choose an index, if the selected index has the value 1 in the centroid vector and 0 in the malicious seed, modify the vector of the malicious seed sample. This cor-



**Algorithm 2 Evasion Attack using permissions (Hamming Distance)**

```

Input: Dataset  $D$ , Testset  $T$ , Classifiers  $H$ , Number of benign samples to be shortlisted  $\beta$ , perturbation limit  $\delta$ 
Output: Evaded Samples
1:  $i \leftarrow 0$  ▷ iteration counter
2: repeat
3:    $x \leftarrow T[i]$  ▷ initialize  $i^{th}$  malware sample vector from T to x
4:    $j \leftarrow 0$  ▷ iteration counter
5:   repeat
6:      $b \leftarrow D[j, 1 : m]$  ▷ initialize  $j^{th}$  benign sample vector from D to b
7:     if  $b[m]=0$  then ▷  $m^{th}$  column represents the class label of a vector
8:        $h \leftarrow \text{hamming\_distance}(x, b)$ 
9:       if  $h \neq 0$  then
10:         $A[j][2] \leftarrow h$  ▷ A is a 2 dimensional array where,  $1^{st}$  column has benign samples  $2^{nd}$  column has the
           distance to x
11:        end if
12:      end if
13:    until  $j \leq |D|$  and
14:    sort A in ascending order of distances
15:     $l \leftarrow A[1 : \beta]$  ▷ l is the 2- dimensional array of benign samples with the shortest distance to malware x
16:     $j \leftarrow 0$ 
17:    repeat
18:       $c \leftarrow 0$  ▷ count of perturbation added
19:       $b \leftarrow l[j]$  ▷ benign vector in A
20:       $a \leftarrow b \text{ XOR } x$ 
21:      select a random number  $\gamma$  s.t.a[ $\gamma$ ] = 1
22:      if  $b[\gamma]=1$  and  $x[\gamma]=0$  then
23:         $x[\gamma] \leftarrow 1$  ▷ adding perturbation
24:         $c \leftarrow c + 1$ 
25:       $P \leftarrow H\_predict(x)$  ▷ testing classifier with evaded sample
26:      if  $p=0$  then ▷ classifier predict it as benign
27:         $i = i + 1$ 
28:        goto 2
29:      else
30:        if  $c < \delta$  then
31:          goto 21
32:    until  $j \leq |l|$ 
33: until  $i \leq |T|$ 

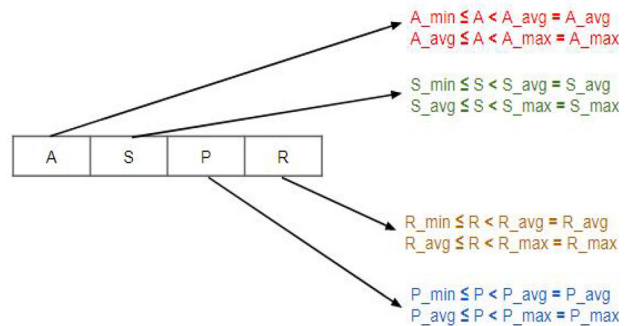
```

responds to the addition of permissions in the malware apk.

6. The new sample with injected permissions are presented to all the classification models. If the models wrongly predict the tainted sample as benign, we select such adversarial samples to perform evasion against the deep neural network.
7. However, if the classification model labels modify samples as malicious, we repeat the process by selecting randomly index of the seed vector. This process is continued until a minimum fraction of permissions is injected into the malicious samples.

**5.3 Evasion attack using app’s components**

App’s components are the basic building blocks of an Android application. The four main app components are Activity, Services, Provider and Receiver. Activities are used for user interaction, Services are an entry point for keeping an app running in the background, the



**Fig. 8** Modification of app components

**Table 1** Statistics of Application components for the legitimate apps

Metrics	Activity	Services	Provider	Receivers
Minimum	0	0	0	0
Average	57	43	24	18
Maximum	130	112	79	53

**Table 2** Comparison between different machine learning classifiers on static, dynamic and hybrid analysis

Method to detect malware	Technique to evaluate predictive models	Classifier	A (%)	F1(%)	P (%)	R (%)
Static analysis	K-fold	LR	96.89	95.61	96.88	94.39
		CART	97.26	96.17	96.45	95.85
		RF	96.41	94.77	99.24	90.70
		SVM	97.59	96.60	97.85	95.40
	Train-test split	LR	96.57	95.25	97.37	93.22
		CART	96.57	95.32	96.05	94.59
		RF	95.67	93.81	99.31	88.88
		SVM	97.10	96.00	97.94	94.14
Dynamic analysis	K-fold	LR	93.00	90.46	88.66	92.37
		CART	93.71	91.52	89.27	93.37
		RF	95.64	94.07	92.21	96.05
		SVM	93.41	90.52	93.67	87.73
	Train-test split	LR	92.77	90.43	88.55	92.39
		CART	93.56	91.47	89.57	93.46
		RF	95.47	93.99	92.17	95.89
		SVM	93.53	90.95	94.21	87.90
Hybrid analysis	K-fold	LR	93.80	91.50	90.20	92.80
		CART	100	100	100	100
		RF	98.54	97.98	98.06	97.90
		SVM	100	100	100	100
	Train-test split	LR	93.19	90.89	89.88	91.93
		CART	100	100	100	100
		RF	98.03	97.31	97.99	96.65
		SVM	100	100	100	100

Receiver helps in delivering events outside the app environment and the Provider manages the shared set of app data. The `AndroidManifest.xml` file contain following tags: `<activity>`, `<services>`, `<provider>` and `<receiver>`. To create samples that can evade classifiers, we count the occurrence of app components defined in the legitimate applications. Figure 8 shows the approach of perturbing malicious apk. In Fig. 8  $A_{min}$ ,  $A_{avg}$  and  $A_{max}$  denote the minimum, average and maximum occurrence of activities in all the benign samples. Similarly  $S_{min}$ ,  $S_{avg}$  and  $S_{max}$  is the minimum, average and maximum number of services in the manifest file,  $R_{min}$ ,  $R_{avg}$ ,  $R_{max}$  denote receiver and  $P_{min}$ ,  $P_{avg}$  and  $P_{max}$  is the estimate of providers declared in goodwill.  $A$ ,  $S$ ,  $R$  and  $P$  are the estimates of activity, services, receiver and provider in a seed malware sample. The number of injected components in a malware seed is either average or a maximum number of specific component appearing in benign applications.

We consider a malware seed sample with 20 activities, 50 services, 35 provider, and 2 receivers respectively. The statistics of the app components in the set of benign applications are shown in Table 1.

Using the approach detailed in Fig. 8, the app components of the malware seed sample are modified. The first feature value  $A = 20$  is in the range  $A_{min} < 20 < A_{avg}$ , hence the activity ( $A$ ) in the seed example is updated to  $A_{avg} = 57$ . The count of services in the seed is altered to  $S = S_{max} = 112$ , as  $S$  is in the range  $S_{avg} < 50 < S_{max}$ . Similarly the old value of  $P = 35$  is updated to  $P_{max}$ , as  $P$  is in the range  $P_{avg} < 35 < P_{max}$ , likewise  $R$  is modified to  $R_{avg} = 18$ . Finally, the seed malware application is augmented with 57 activities, 112 services, 24 providers, and 18 receivers. If the modified app is wrongly labelled by the classification models, then a set of such samples have the potential to deceive detection. Otherwise, we increment the count of each component by a value of 3 until the modified app is miss-classified by the classification models.

## 6 Experimental evaluation

The study consists of two experiments. The purpose of the first experiment was to compare the performances of classifiers trained with features obtained with static, dynamic, and

**Table 3** Comparison of the results obtained for static, dynamic, and hybrid analysis based on Deep Learning

Method to detect malware	A (%)	P (%)	R (%)	F1(%)
Static analysis	99.28	98.99	99.08	99.04
Dynamic analysis	94.61	90.54	95.51	92.96
Hybrid analysis	99.59	99.63	99.27	99.45

hybrid analysis. The second experiment aims at evaluating how the performances of classifiers degrade when subjected to the adversarial examples.

## 6.1 Dataset and experimental setting

For the first experiment we consider, 5,694 benign applications, and 3,197 malware applications. The benign applications were downloaded from the Android App store “9apps”. The Drebin dataset [7] is considered as the malware dataset as it is widely used for experiments and testing of malware classifiers and detectors. Subsequently, in the second experiment for evaluating the robustness of the machine learning and deep learning models, we augmented both malware and benign dataset retaining apks from the first experiment. A total of 11, 447 applications comprising 6,072 benign apks (from 18 different categories) and 5,375 malware apks were collected. Employing VirusTotal<sup>1</sup> we accepted as benign those apps that were labelled as goodware by the majority of antivirus offered by VirusTotal.

All our experiments were conducted on a system with an i7 processor, 8GB RAM, 256 SSD and, 1TB HDD, running the 64-bit Ubuntu operating system. The software requirements were Android Studio and, Anaconda. Anaconda Python distribution was used to execute machine learning in Python language with the help of libraries Scikit-learn, Keras, Matplotlib. Classifiers used in this study are logistics Regression, Random Forest, Support Vector Machine and Deep Neural Network. Hyperparameters for classifiers are tuned using a random search method.

## 6.2 Evaluation metrics

The metrics used for evaluating the performance of the classifiers are accuracy, the F1, precision and recall. Malware classified as malware represents the True Positive ( $TP$ ), malware classified as benign represents False Negative ( $FN$ ), benign app classified as malware represents False Positive ( $FP$ ) and benign application classified as benign app represents True Negative ( $TN$ ). Accuracy, precision, recall and

$F1$  are defined with the following equations.

$$Accuracy(A) = \frac{TP + TN}{TP + FP + TN + FN} \quad (1)$$

$$Precision(P) = \frac{TP}{TP + FP} \quad (2)$$

$$Recall(R) = \frac{TP}{TP + FN} \quad (3)$$

$$False\ Positive\ Rate(FPR) = \frac{FP}{FP + TN} \quad (4)$$

$$F1\ score(F1) = 2 * \left( \frac{P \times R}{P + R} \right) \quad (5)$$

## 6.3 Results of experiment-I

**Static Analysis** :In static analysis, the attribute length in the experiments carried out is 3,360. The highest accuracy and  $F1$  was observed for the SVM classifier, even if the best precision is obtained by RF in k-fold and in train-test split, while recall is better for CART classifier in both k-fold and train-test split.

**Dynamic Analysis** :In dynamic analysis, the attribute length is 2,425. It is observed that RF produced the highest accuracy,  $F1$  and recall compared to LR, CART and SVM classifier, even if the precision is greater for SVM classifier in k-fold and 2.04% in train-test split.

**Hybrid Analysis** :In hybrid analysis, the feature length is 5,785. It is observed that CART and SVM classifier obtained the highest accuracy,  $F1$ , precision and recall: we can conclude that the hybrid features provide the highest performances.

Using Fischer score prominent attributes were selected to obtain variable feature vector comprising of 10%, 20%, 30%, 40% and 50% of original feature space (which is 3,360, 2,425 and 5,785, respectively as discussed above). Table 2 reports comparison between different machine learning classifiers on static, dynamic and hybrid analysis. Specifically we report the results of k-fold cross-validation and train-test split approach to evaluate predictive models.

### 6.3.1 Performance of deep neural network

The conventional machine learning algorithms accurately detect unknown samples if specialised feature engineering methods are put in place for extracting attributes representative of target classes. Thus, it demands discovering attribute selection methods that can capture the behaviour of the applications capable of categorizing samples into a specific class. Usually extracting a subset of features from a feature space by applying diverse feature selection approaches is time-consuming. Even if a set of significant attributes are derived, the next challenge is the adoption of a suit-

<sup>1</sup> <https://www.virustotal.com/gui/>.

**Algorithm 3 Evasion attack on permission:K-Means clustering****Input:** Dataset  $D$ , Test set  $T$ , Classifiers  $H$ , Number of clusters  $\rho$ , the threshold for sigmoid function  $f$ - $\mathfrak{S}$ , perturbation limit  $\delta$ **Output:** Evaded Samples

```

1: procedure K-Means clustering (Dataset  $D$ )
2:   initially choose  $\rho$  data points from  $D$  as centroids
3:   (re)assign each vector in  $D$  to the cluster to which it is closer relying on the mean value of the object in the cluster
4:   update the cluster means
5:   centres  $\leftarrow$  cluster_means
6: end procedure
7:  $i \leftarrow 0$  ▷ iteration counter
8: repeat
9:    $c \leftarrow$  centers[ $i$ ] ▷ initialize  $i^{th}$  centroid vector from centers to  $c$ 
10:   $j \leftarrow 0$ 
11:  repeat
12:     $s \leftarrow f[c[j]]$  ▷  $f$  is the sigmoid function applied on each value in  $j^{th}$  centroid
13:    if  $s > T$ 
14:       $c[j]=1$ 
15:    else
16:       $c[j]=0$ 
17:       $j = j + 1$ 
18:    until  $j \leq |c|$ 
19:     $i = i + 1$ 
20: until  $i < \rho$ 
21:  $i \leftarrow 0$  ▷ iteration counter
22: repeat
23:   $x \leftarrow T[i]$  ▷ initialize  $i^{th}$  malware sample vector from  $T$  to  $x$ 
24:   $j \leftarrow 0$  ▷ iteration counter
25:  repeat
26:     $c \leftarrow$  centers[ $j, 1 : m$ ] ▷ initialize  $j^{th}$  centroid  $v$ 
27:     $a \leftarrow c$  XOR  $x$ 
28:    select a random number  $\gamma$  s.t.  $a[\gamma] = 1$ 
29:    if  $b[\gamma] = 1$  and  $x[\gamma] = 0$  then
30:       $x[\gamma] \leftarrow 1$  ▷ adding perturbation
31:       $c = c + 1$ 
32:       $P \leftarrow H\_predict(x)$  ▷ testing classifier with evaded sample
33:      if  $p = 0$  then ▷ classifier predict it as benign
34:         $j = j + 1$ 
35:        goto 24
36:      else
37:        if  $c < \delta$  then ▷ check if number of perturbations added is beyond the limit
38:          goto 27
39:        end if
40:      end if
41:       $j = j + 1$ 
42:    until  $j \leq |l|$ 
43:   $i = i + 1$ 
44: until  $i \leq |T|$ 

```

able approach for representing applications, in particular feature vector representation. Both the aforesaid techniques, i.e., feature engineering and attribute representation require domain-specific knowledge. The dark side of such a proposal for security systems is the threat of adversarial attacks affecting the integrity and availability of such malware scanners.

To overcome the limitations posed by conventional machine learning algorithms, deep learning neural network models are used as an extension in this study. The primary objective is to improve the detection of malicious apks without the need of implementing feature selection and representation. Thus, we developed three DNN models for predicting samples by using attributes such as (a) permissions (b) sys-

tem calls and (c) a combination of permissions and system calls. Further, before deploying the classification models for predicting apks, hyper-parameters were tuned. In particular, we investigated fixing the best optimizer from a collection of optimizers (rmsprop, adam) and initializers from a collection of initializers (glorot\_uniform, uniform). Additionally, we tuned drop-out rate, epochs and batch size. Further, speeding the search of optimal hyper-parameters GridSearchCV approach was adopted. The number of epochs, batch size, and the dropout rate is different in all three models. A small description of these parameters and their values are discussed below.

The dataset has to be propagated forward and backwards through the neural network and this denotes one epoch. But it is too large to pass the entire dataset in one epoch. So it is divided into smaller batches. In the initial static analysis model, the number of epochs is 50 and its batch size is set to 500. In the dynamic analysis model, the number of epochs is raised to 250 and its batch size is reduced to 200. In the hybrid analysis model, the number of epochs is 150 and its batch size is set to 300.

Dropout is a technique used to reduce overfitting, which randomly ignores some layer's output. In the static analysis model, its rate is 0.0, which denotes no outputs from the layer. For both dynamic and hybrid analysis models, it is 0.4. That is, 40% of the neurons in the neural networks are ignored.

Table 3 reports the results obtained for static, dynamic, and hybrid analysis based on deep learning. The static analysis model using deep learning has the highest accuracy, precision, recall, and  $F1$  compared to the highest performance static analysis SVM model based on machine learning. That is, accuracy, precision, recall,  $F1$  is increased by 1.69%, 1.14%, 3.68% and 2.44%. The machine learning-based RF model has a better performance compared to the deep learning-based model for dynamic analysis. That is, accuracy is greater by 1.03%, precision is greater by 1.67%, recall is greater by 0.54%, and  $F1$  is greater by 1.11%.

Finally, in hybrid analysis, the machine learning-based CART and SVM models exhibit higher accuracy, precision, recall, and  $F1$  compared to the deep learning-based model. That is, accuracy is higher by 0.41%, precision is higher by 0.37%, recall is higher by 0.73%, and  $F1$  is higher by 0.55%. However, comparing the results of static, dynamic, and hybrid models using deep learning, the hybrid model has the highest performance. This again shows that hybrid models can exhibit better results than standalone static and dynamic models.

### 6.3.2 Comparative analysis

The proposed system that uses multi-modal features, i.e. hybrid features is compared with the following solutions developed on the same dataset

Surendran et al. [42] proposed GSDroid, which leverages graphs for representing system calls sequence extracted from applications in lower-dimensional space. Experiments were conducted on 2,500 malware and benign samples. Malware applications included 1,250 apps from Drebin and the same number of goodware downloaded from Google Playstore. GSDroid reported 99.0% accuracy and  $F1$ . Bernardi et al. [9] adopted an approach based on model checking for detecting Android malware on 1,200 apk's from Drebin dataset. They created a system calls execution fingerprint (SEF); the obtained SEFs were given as an input to the classifier, reporting 0.94 as True Positive Rate. Finally, SAMADroid [8] is

a 3-level malware detection system that operates on a local host and remote server. Random forest model trained on static features resulted in 99.07% accuracy. However, through our solution based on hybrid features, the accuracy of DNN and SVM is 99.59% and 100% respectively which is far better than the solutions discussed above.

### 6.3.3 Execution time

The time for detecting samples in our system can be measured based on the time consumed in each module. Here, we discuss the time expended for extracting system calls. Each application is executed for 60 seconds in an emulator, with 200 random events generated by Android Monkey. Overall an average of 92 seconds is required for the entire operation, which comprises booting a clean virtual machine, installing the app, generating the system call logs, copying logs to the host and finally reloading fresh VM. After extracting features, we created a data structure known as the feature vector table (FVT), which is a collection of the feature vectors. We represent the feature space as a binary tree that requires  $O(\log n)$ . FVT is presented to the classification algorithms for building classifiers. Finally, training Random forest, SVM, CART, LR and DNN requires 5,296 ms, 4,750 ms, 4,076 ms, 899 ms and 6,322 ms respectively.

## 6.4 Experiment-II: performance of classifiers on adversarial examples

In the following section, we discuss the performance of classifiers presented with adversarial samples. These evasive applications are developed by injecting additional permissions and app components. Additionally, we report the attributes responsible for transforming malware apk's to legitimate applications.

### 6.4.1 Adversarial applications developed with similarity measure

Table 4 shows the performance of different classification models. It can be seen that  $F1$  for predicting applications in the test set is in the range of 0.964-0.970. We randomly selected 537 malicious applications from the test set and determined the similarity with legitimate applications. Extra permissions absent in malware samples but present in the benign dataset were added to these malicious applications. After submitting such tainted (adversarial) applications, the average detection rate and false-positive rate of classifiers obtained are 44.13% and 55.86% respectively. Overall 300 tainted malware samples were created from 537 malware seed samples by merely altering permissions identical to 0.5% of benign applications.

**Table 4** Performance of classifier on Adversarial Examples developed using Hamming Distance

Training Set Classifiers	A	F1	P	R
<i>Before Attack</i>				
LR	0.964	0.958	0.975	0.943
RF	0.964	0.958	0.987	0.930
SVM	0.965	0.960	0.975	0.946
<i>Test Set</i>				
LR	0.937	0.967	1.0	0.937
RF	0.931	0.964	1.0	0.931
SVM	0.942	0.970	1.0	0.942
FNR	TPR	#Evaded sample	Mean attributes	Standard
<i>After Attack</i>				
55.86%	44.13%	300	altered 7.02	deviation 6.108

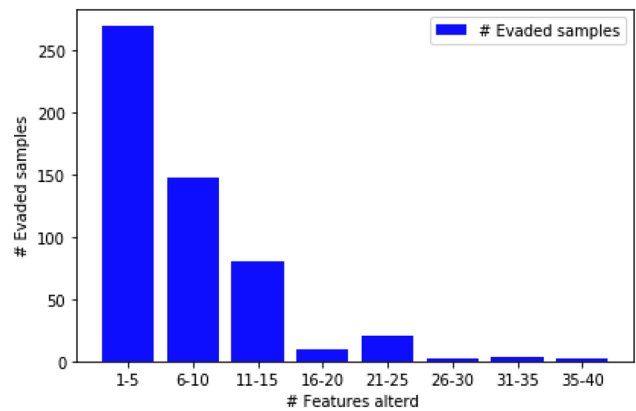
**Table 5** Permission-based attack on Deep Neural Network, adversarial examples have high similarity (Hamming distance) with the legitimate applications

Dropout	A(%)	F1(%)	P(%)	R(%)
<i>Before Attack</i>				
0.6	98.38	98.25	99.08	97.32
FNR(%)	A(%)	F1(%)	P(%)	R(%)
<i>After Attack</i>				
45.94	51.62	68.02	1.0	51.62

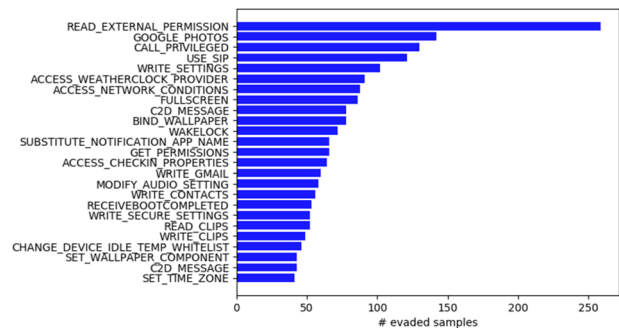
Similarly we simulated an identical permission-based attack on a deep neural network. In this way, the statistics of permissions in adversarial samples should be close to legitimate applications. The results in Table 5 show a decrease in F1 (68.02%) after the attack, consequently an increase in 45.94% of False Negative Rate is obtained. Overall, 300 malware samples in the test set evaded the detection by merely changing 38 permissions in the malicious applications.

The distribution of evaded malware samples is shown in Fig. 9. It is seen that 50.27% malicious samples (270 nos.) can bypass DNN by solely changing 1 to 5 permissions, 27.5% adversarial samples evade detection by altering 6 to 10 features. As opposed to this, 2 to 4 samples require the addition of 20 permissions to escape detection.

In Fig. 10, we show permissions that are frequently inserted majorly in adversarial samples. In particular, we show the top 25 permissions injected in malware applications through which they escape detection.



**Fig. 9** Number of evaded samples vs number of permissions inserted



**Fig. 10** Inserted permissions in adversarial samples

### 6.4.2 Adversarial applications generated by estimating the similarity with clusters of goodwill

In the previous scenario, the similarity of malware applications reserved for generating adversarial samples (A) is computed with all benign applications (B) which were not part of the training set. The overall computational cost of

estimating similarity using Hamming distance (discussed in Sect. 6.4.1) is  $O(\mathcal{A} \times \mathcal{B})$ . In this experiment, using the K-Means clustering approach, we create  $\rho$  clusters of benign samples ( $\mathcal{B}$ ). Now the distance of each malware sample in  $\mathcal{A}$  is computed with benign applications in  $\rho$  centroids, hence the complexity is  $O(\mathcal{A} \times \rho)$  which is less than  $O(\mathcal{A} \times \mathcal{B})$ . The centroids are real-valued vectors. As the feature vectors are in binary form, the centroids are converted to binary-valued vectors using a sigmoid function. The threshold  $\tau$  for the sigmoid function is considered. If a value of the sigmoid function is greater than  $\tau$ , then the number is mapped to 1 otherwise is retained as 0. For example, let us consider centroid of a cluster as [3.21, 5.13, 0.77, 6.71, 2.54, 1.78, 7.89, 4.62], and the threshold is assumed as 0.5. Thus, centroid is transformed to binary vector as [0, 1, 0, 1, 0, 0, 1, 0]. In this work, the  $\tau$  is in the range of 0.5 to 0.6 obtained in increments of 0.02. Experiments are conducted for different cluster size, i.e.,  $k = 3, 5$  and 10, shown in Table 6.

From Table 6, 100% evasion of adversarial samples are obtained at threshold of 0.5 for  $k = 3, 5$  and 10. The average number of permissions inserted for cluster size  $k = 3$  is higher compared to  $k = 5$  and 10. Further, we observe as threshold increases the average percentage of evasions decreases.

### 6.4.3 Evaluation of poisoning attack on app components

In this scenario we randomly chose 537 malware applications from the test set and injected different components. The results obtained is shown in Table 7 and Table 8. The highest  $F1$  and accuracy is obtained with Random forest, all other classifiers report poor accuracy. One of the fundamental reason is the lack of attributes to separate applications of target classes. Generally, DNN needs a large number of features to extract relevant attributes to perform precise prediction of the presented samples. Thus, we see that the highest accuracy of 78.1% is obtained with a deep neural network which justifies the lack of attributes for classification. Also, we observe that merely increasing the number of app’s components in the malicious application can easily deceive machine learning and deep learning classifier. In particular, the increase in the frequency of a particular component changes the direction of classification and the learned hypothesis function cannot appropriately predict the new applications.

### 6.4.4 Attacks using system calls

In this section, we create adversarial examples (AE) using system calls to launch evasion attack (where the attacker aims to affect the target model) and poisoning attack (adversary has the access to training data, to influence model performance). We simulate attacks on a set of machine learning and deep learning models. For deceiving models, partic-

**Table 6** Adversarial examples created using k-means clustering

Threshold	Avg. Attributes altered (%)	Evasion (%)	FNR (%)	TPR (%)
<i>No. of Cluster (k = 3)</i>				
0.5	55.1	100	100	0
0.52	0.95	87.15	87.15	12.84
0.54	1.14	77.74	79.08	20.91
0.56	1.33	84.91	96.64	3.35
0.58	1.38	74.23	85.72	14.27
0.6	1	38.36	38.91	61
<i>No. of Cluster (k = 5)</i>				
0.5	12	100	100	0
0.52	0.84	90.8	98.92	9.66
0.54	0.84	73.03	76.05	33.81
0.56	0.84	73.7	78.69	21.3
0.58	1.38	45.47	48	14.27
0.6	0.707	54.45	67.03	32.9
<i>No. of Cluster (k = 10)</i>				
0.5	6.47	1	1	0
0.52	0.89	75.34	90.33	9.66
0.54	0.92	59.01	66.18	33.81
0.56	0.51	40.94	46.03	53.96
0.58	0.95	37.7	43.66	56.33
0.6	1	48.41	64.67	35.32

**Table 7** Performance of classifier on evasive malware variants injected with app components

Training Phase Classifier	A (%)	F1(%)	P(%)	R(%)
<i>Before Attack</i>				
LR	75.86	76.14	66.14	89.7
RF	86.42	84.76	81.78	87.96
SVM	81.97	78.16	81.44	75.13
<i>Testing Phase</i>				
LR	89	94.18	100	89
RF	88.1	93.67	100	88.1
SVM	74.05	85.09	100	74.05
<i>After Attack</i>				
FNR	TPR	Evaded sample		
90.13%	98%	484		

**Table 8** Performance of DNN on evasive malware variants injected with app components

Drop out	A	F1	P	R
<i>Before Attack</i>				
0.5	78.21%	79.98%	68.75%	95.6%
<i>After Attack</i>				
100%	0%	0%	0%	0%

ularly SVM, Random forest, dense neural networks and 1D-Convolutional Neural Network (1D-CNN). The detailed configuration deep neural network (DNN) and 1D-CNN is presented in Table 9. We assume that the attacker has partial knowledge about the system, in this context the classification algorithms. However, the attacker has access to alternate malware dataset from public repositories. With these capabilities, the adversary is capable of deriving discriminant features and use a subset of attributes to create evasive malware variants. In particular for simulating this form of attack, discriminant attributes from the training set are obtained employing *SelectKBest (SK)* and *Recursive Feature Elimination(RFE)* methods from `sklearn.feature_selection` module. Moreover, for each app,  $n$ -gram profiles are created, then each file is represented as uni-gram and bi-grams of system calls.  $n$ -grams have been extensively studied in malware detection [37] [1], and have proven to efficiently identify malicious samples from a collection of large examples consisting of both malware and goodware. Figure 11 provides the difference in the distribution of  $n$ -grams (system call grams) in malware and benign applications.

Before applying attribute selection methods, we trimmed the feature space by eliminating  $n$ -grams with a score less than or equal to 0.0001. Later, features are further synthesized using *SelectKBest* and *Recursive Feature Selection*. In the case of uni-gram 96 system calls are reduced to 83, and finally, 56 uni-grams are extracted through feature selection methods. Likewise, out of 2,364 bi-grams, 166 call grams are chosen using the threshold and finally, 83 call sequences are obtained with attribute selection methods.

We performed the prediction on 10% of randomly selected malware samples ( $T$ ) excluded from the training set by appending discriminant system calls. We set the maximum attack iteration ( $I_{max}$ ) to 30%, which means discriminant system calls are repeated at the end of each sample  $\tau \in T$  which satisfies the condition that  $|\tau| + gram \leq I_{max}$ . To evaluate the efficacy of the evasion attack we measured the amount of system call gram added to each file  $\tau$ : the percentage of calls appended to the file is in the range of 5%-30%, while the inserted ones in increments of 5%.

### (A) Evasion attacks using system call

We performed the experiments with 247 randomly selected malware samples as the test set (10% of applications). Figure 12 provides the results attained by progressively appending system calls to the samples in the test set. Before the attack, the F1-measure of uni-gram models (SVM-SK, SVM-RFE, RF-SK AND RF-RFE) are 0.952, 0.950, 0.981 and 0.99 respectively. A significant drop in F1 is observed for each model (refer Fig. 12a) by adding 5% of system calls to each file in the test set. Overall, F1 of the model after the attack is observed between 0.10 to 0.15.

While in the case of bi-gram model, F1 score for the above mentioned classifiers are in range of 0.961 to 0.988 (also shown as 0% in Fig. 12b). We see a marginal drop in F1 for RF-RFE model and a maximum overall drop of 1.6% after the attack. Notably, adding call sequences to uni-gram models is effective compared to bi-gram ML models. We also observe that RF-RFE model trained on RFE features can withstand an evasion attack. RFE being a wrapper-type feature selection algorithm utilizes a classification algorithm to measure the importance of attributes. As the stability of RFE depends primarily on the wrapper(classification algorithm), thus relatively improved outcome is obtained with Random Forest (RF). The superior performance of Random Forest is attributed to the fact that the relevant attributes are filtered by bootstrapping the samples and features. In this way, several decision trees are created which contribute to computing the model performance.

Figure 12(c) present the results of Deep neural network (DNN) and 1D-CNN on evasion attack. For DNN F1 drops from 0.967 to 0.375 and 0.562 respectively adding extra 5% system calls in each malware samples in the test set. The classifier performance is severely affected by increasing the number of system calls being added to files. Here, we observe that a significant misclassification is obtained, however, the rate of misclassification for bi-gram models are comparably less than models trained on uni-grams. Additionally, we evaluated the robustness of 1D-CNN; results are shown in Fig. 12d. The evaluation was conducted on variable stride length which can be considered as  $n$ -grams. Before the attack, the F1 scores on distinct strides are 0.9788, 0.981 and 0.9815 respectively. However, after the evasion attack malware samples were wrongly labelled as legitimate, thus the drop in F1 by padding 5% discriminant system calls to each file are 5.88%, 2.06% and 2.75% respectively. On comparing individual models, it can be seen that the 1D-CNN offer higher resistance to evasion attacks. 1D-CNN can derive robust features without the use of a complex feature engineering process, and have a computational complexity of  $O(K.N)$ , where  $K$  is the kernel and  $N$  is the size of the input.

### (B) Poisoning attack using system call

In the following paragraphs, we discuss the evaluation of the poisoning attack. We simulate the behaviour of an adversary



**Table 9** Configuration of DNN and 1D-CNN

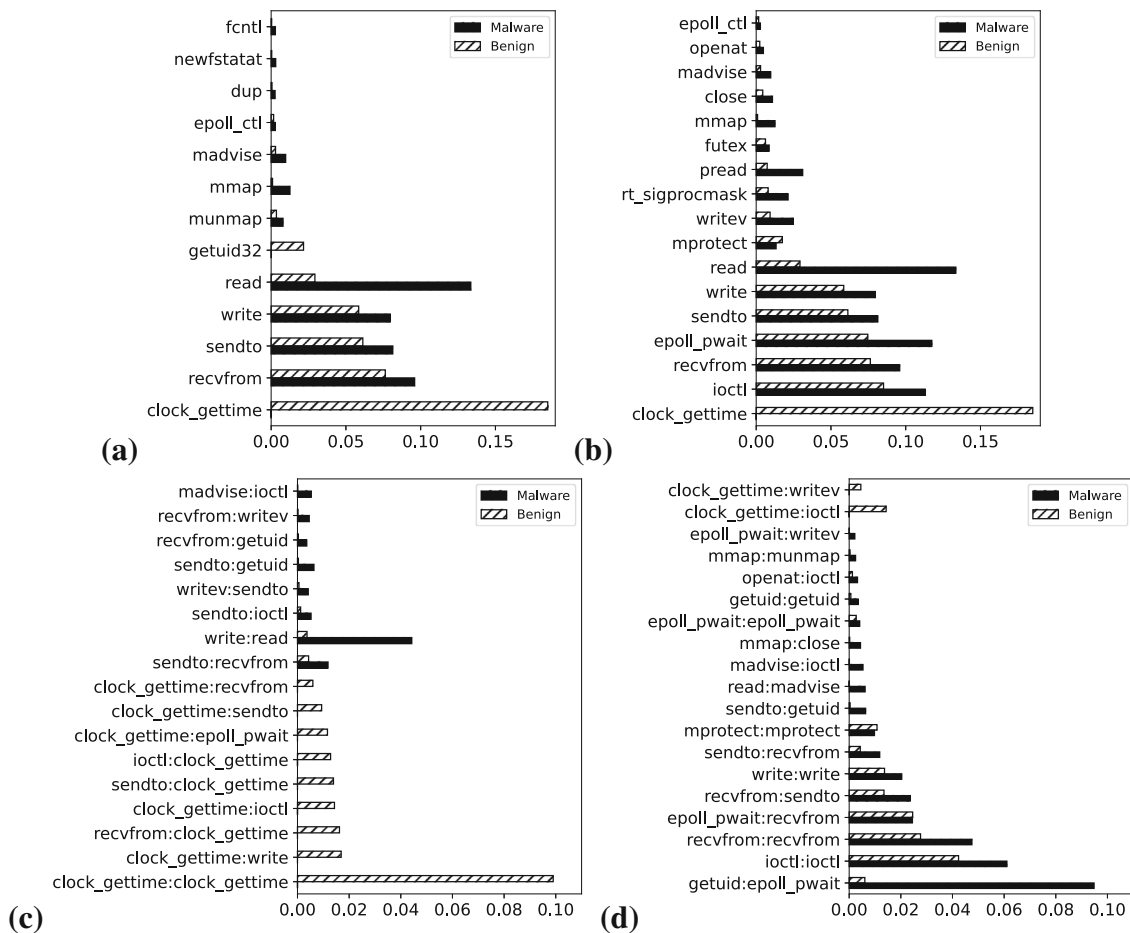
Model	Input	Layers	Hyperparameters
DNN (Uni-gram)	96	<b>Layer - 1 (Hidden)</b> Dense(128) + Dropout(0.1) + BatchNormalization <b>Layer - 2 (Hidden)</b> Dense(256) + Dropout(0.2) + BatchNormalization <b>Layer - 3 (Hidden)</b> Dense(512) + Dropout(0.3) + BatchNormalization	Learning rate = 0.0001 Epochs = 100, Batch size = 16 Optimizer = Adam Hidden layer activation = Relu Output layer activation = sigmoid
DNN (Bi-gram)	2364	<b>Layer - 1 (Hidden)</b> Dense(64) + Dropout(0.1) + BatchNormalization <b>Layer - 2 (Hidden)</b> Dense(32) + Dropout(0.2) + BatchNormalization <b>Layer - 3 (Hidden)</b> Dense(16) + Dropout(0.3) + BatchNormalization	Learning rate = 0.0001 Epochs = 50, Batch size = 16 Optimizer = Adam Hidden layer activation = Relu Output layer activation = sigmoid
1D-CNN (Stride 1 -3)	101681	<b>Layer - 1 (Embedding)</b> Embedding(32) <b>Layer - 2 (Hidden)</b> Conv1D(128) <b>Layer - 3 (Hidden)</b> MaxPooling1D <b>Layer - 5 (Hidden)</b> Conv1D(256) <b>Layer - 6 (Hidden)</b> MaxPooling1D <b>Layer - 7 (Hidden)</b> Conv1D(512) <b>Layer - 8 (Hidden)</b> MaxPooling1D <b>Layer - 9 (Hidden)</b> Dense(10)	Learning rate = 0.0001 Epochs = 30, Batch size = 8 Optimizer = Adam Kernel size = 3 Hidden layer activation = Relu Output layer activation = Sigmoid

who manipulates a subset of malware files in the training set by appending a set of selected system call sequence (extracted using feature selection methods). The overall objective is to maximize the classifier confidence in labelling malicious file as legitimate, or in other words, increase the probability of tainted samples classified as benign. An alternate scenario of poisoning attack is the label flipping attack, here the adversary deliberately swaps the original label of a sample with the target class label. In our study we focused on developing poisoned samples by adding extraneous system call to selected malware seed samples. Figure 13 presents the results of poisoning attack.

Practical use case of poisoning attack in malware detection domain is crowd-sourcing the malware apps for labelling and generating its signatures. Under such circumstances, a dishonest user can manipulate the samples or intentionally modify the label. However, the attack can be defeated in the presence of a large number of legitimate users, where the

class label of a suspect file is decided relying on majority voting. Mimicking such a scenario we intended to poison a very small fraction of malwares in the training set. Figure 13(a) provides the outcome of ML classifiers on padding uni-grams. We observe here that a small fraction of samples in the test set is misclassified. The overall drop in average F1 for the RF-RFE and RF-SK is 0.068%, 0.25% respectively. Likewise, in the case of SVM-SK and SVM-RFE the average drop in F1 are 3.32% and 1.596%. We can conclude that Random forest models are highly resistant to adversarial attack, specifically, the performance of RFE trained models show improved results with respect to the models trained on *SelectKbest* attributes.

Similar trends in the results are obtained for bi-gram models (refer Fig. 13b. For SVM-SK classifier the difference in F1 falls in the range of 0.004 to 0.006 compared with the model in the absence of a poisoning attack, where the F1 is 0.963. In the case of SVM-RFE the average change in F1 for the entire



**Fig. 11** System call grams **a** uni-gram SelectKBest **b** uni-grams RFE **c** bi-gram SelectKBest and **d** bi-gram RFE

range of padded system calls (i.e., 5% to 30%) is 0.575% with the standard deviation of 0.0006. A very small spread in the F1 values indicates the ineffectiveness of poisoning attacks. Identical observations can be made for Random forest models (RF-RFE and RF-SK), where the spread of F1 across a different range of padding is 0.00035 and 0.00031 respectively.

Figure 13(c) and (d) show the performance of DNN and 1D-CNN. It is evident from these figures that the attack is not severe, and a marginal drop is observed when malware samples are padded with system calls in a larger amount. However, a clear trend is not noticed in the case of deep learning models. Training set with tainted samples in certain cases also improves the classifier results. On investigating the confusion matrix we found that for larger padding size malware samples that were previously misclassified were now precisely detected by DNN. It is intuitive that malicious data points statistically closer to the legitimate files are now accurately detected.

## 7 Evaluation on obfuscated samples

Software developers obfuscate the source code of applications to avoid manual analysis and violations of intellectual property. Instead, malware writers use obfuscation to keep new variants of original malicious applications being detected. A vast majority of malware variants have less than 2% difference in code [22]. Anti-malware products employing pattern matching techniques fail to detect obfuscated files. By forcing an application to execute in an emulated environment, and monitoring system call invocation, obfuscated samples are identified. To generate obfuscated malware variants, we make use of an open-source obfuscator known as Obfuscapk [5]. Obfuscapk supports obfuscation techniques like trivial, renaming, encryption, code reorder and reflection. As the first step, we looked at detecting obfuscated samples in the dataset. In this step, we represented system call invocation of a file as a system call co-occurrence matrix of size  $m \times m$ , where  $m$  is the number of unique calls. Each element in the matrix corresponds to the occurrence of a pair of calls. The call frequencies are normalized and mapped to pixels

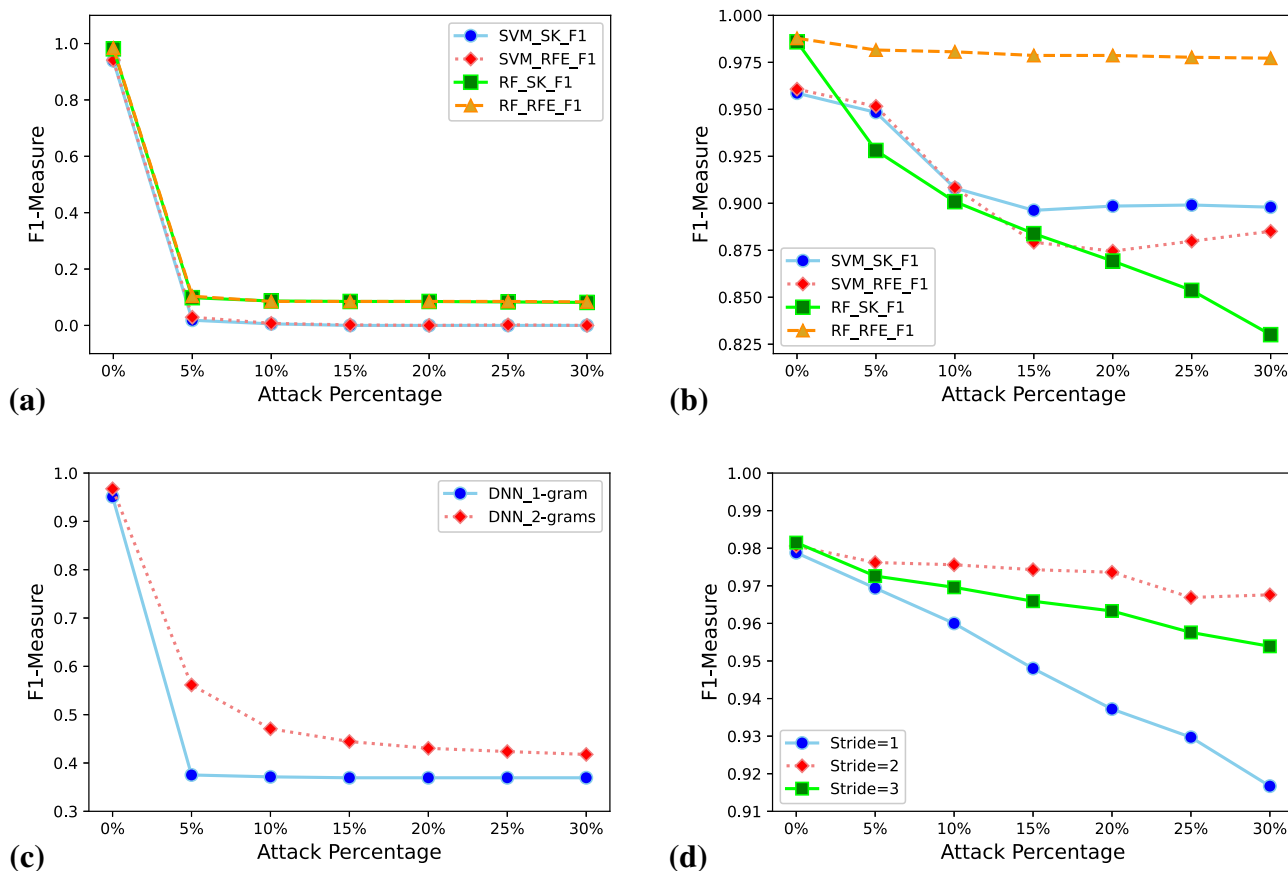


Fig. 12 Evasion attack using a uni-gram SelectKBest b bi-gram RFE c Deep neural network and d 1D-CNN

Table 10 Evaluation of obfuscated malware using system call images

Approach	Accuracy	Precision	Recall	F1	Time
Train-Test	0.966	0.936	0.980	0.957	27sec
CV	0.960	0.928	0.972	0.949	33 sec

by multiplying the normalized values with 255. Finally, the system call images corresponding to malware and benign set are used for training the 2D-CNN model for prediction. We chose CNN for developing the model as it extracts relevant patterns in images even if they are not fixed. To be precise, CNN is spatially invariant to patches of a given image. This is fundamental to code obfuscation where the blocks of code in the program are randomly rearranged by the obfuscator using branch instructions. Table 10 shows the identification of obfuscated malware using *train\_test\_split* and stratified ten-fold cross-validation (CV) approach.

We can observe that the highest F1 obtained by transforming apps into a system call co-occurrence matrix is 0.959. Analysis of co-occurrence matrix revealed the presence of a large number of contiguous blocks of black regions indicating the existence of zeros in this matrix. To improve the

detection, we addressed the problem by transforming malware as gray-scale images, similar to the approach in [31]. In this context, we map raw bytes of .dex files to pixels and apply image processing techniques. Initially, we investigated training ML models on images, especially on image textures extracted using a bank of Gabor filters formed by varying the kernel size, standard deviation, angle, wavelength and aspect ratio. As the feature extraction and training was computationally expensive, we considered employing 2D-CNN, which extracts features without manual intervention from raw malware binaries. For retaining the semantic information of an image, pairwise probability of bytes(pixels) were estimated. Subsequently, the probabilities are transformed into pixel values between 0-255. As a consequence, each apk is converted to a fixed size image (256×256). We train tuned Convolutional Neural Network (CNN) (learning rate = 0.0001, momentum = 0.9, epoch = 100 and batch size = 32) on the generated images of malware and benign samples. The topology of the network is presented in Table 11.

Malware samples used in the previous experiments (refer Section 6.1) [7] are obfuscated, and the performance of the CNN model is estimated under four scenario (a) malware ( $\mathcal{M}$ ) vs benign ( $\mathcal{B}$ ) (b) benign ( $\mathcal{B}$ ) vs obfuscated malware

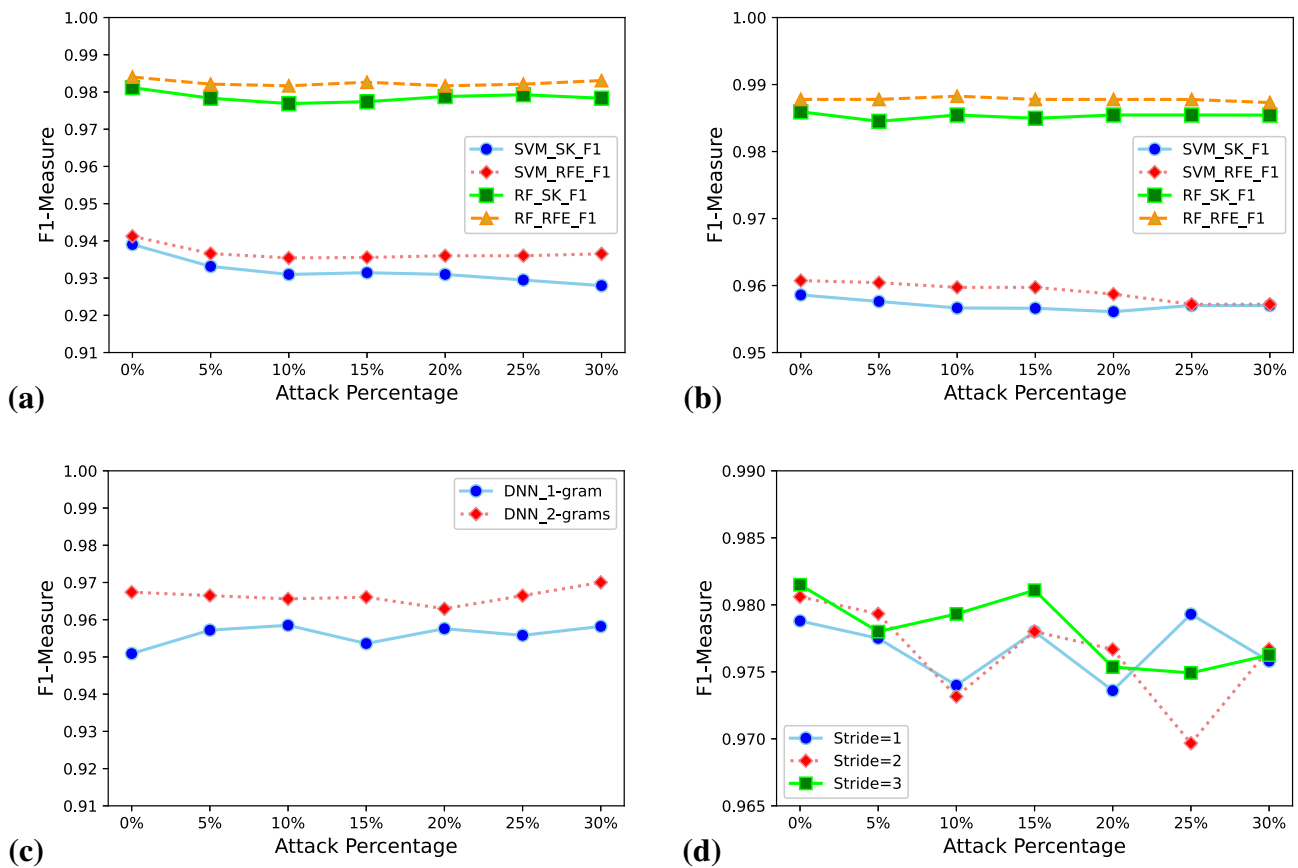


Fig. 13 Poisoning attack employing a uni-gram SelectKBest b bi-gram RFE c Deep neural network and d 1D-CNN

Table 11 Architecture of CNN

Layers	Filter size	Input Shape	Output Shape	Activation
Conv-1	64(3*3)	(64,64,1)	(none,62,62,64)	ReLU
MaxPooling-1	(2*2)	(none,62,62,64)	(none,31,31,64)	-
Conv-2	64(3*3)	(none,31,31,64)	(none,29,29,64)	ReLU
MaxPooling-2	(2*2)	(none,29,29,64)	(none,14,14,64)	-
Dense-1	(none,128)			ReLU
Dense-2(binary)	(none,1)			Sigmoid
Dense-2(categorical)	(none,14)			Softmax

( $\mathcal{M}^\perp$ ) (c) malware( $\mathcal{M}$ ) vs obfuscated malware( $\mathcal{M}^\perp$ ) and (d) malware family class ( $\mathcal{FC}$ ). Figure 14 shows the classification of obfuscated malware family classification. Through this experiment we conclude that CNN accurately labels each sample in the test set to the appropriate obfuscation class. Table 12 presents the results obtained using 2D-CNN.

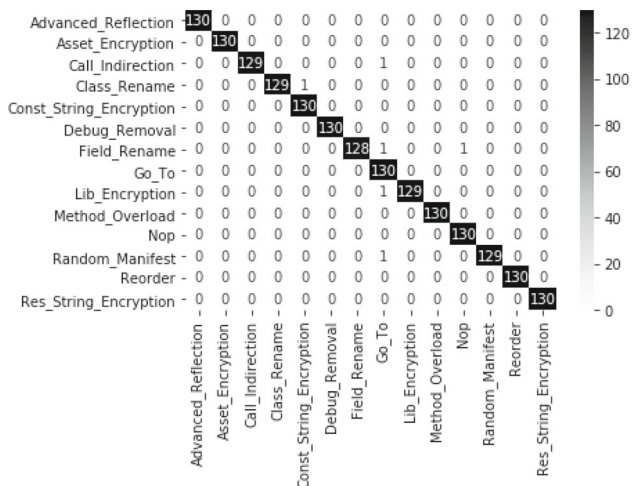
### 8 Discussion

In this study, we show that machine learning classifiers are vulnerable to adversarial attack. ML-based Malware detectors trained on static features such as permissions, APIs

and applications components can be easily attacked by carefully generating perturbed apps having statistical similarity with legitimate apps. Generally, the vector corresponding to an application is represented with boolean values. Iterative addition of features (permission, hardware feature and intents, etc) generates evasive applications with minimal effort without compromising app functionality. In this context, an attacker must modify selected attributes with a value 0 to 1. Further, changing minimum subset of attributes will force linear classifier such as logistic regression, SVM (linear kernel) to misclassify files in the test set. However, significant attempts are required to bypass the classifier trained with the sequence of system calls, as values of features are continu-

**Table 12** Performance of CNN using different proportion of training and test set

Data Split	$\mathcal{M}$ vs $\mathcal{B}$		$\mathcal{B}$ vs $\mathcal{M}^\perp$		$\mathcal{M}$ vs $\mathcal{M}^\perp$		$\mathcal{FC}$	
	A	F1	A	F1	A	F1	A	F1
70:30	0.996	0.995	0.987	0.989	0.997	0.996	0.997	0.997
80:20	0.994	0.994	0.995	0.996	0.998	0.998	0.996	0.996
90:10	0.995	0.995	0.995	0.990	0.999	0.999	0.997	0.996



**Fig. 14** Classification of obfuscated malware variants

ous. This require padding of larger amount of discriminant calls sequence to each malware sample. Intuitively it means that the modified applications will spend large execution time compared to its normal functionality. It is worth mentioning that such suspicious apps will be easily detected by monitoring the power consumption and heat dissipated of the smart device. Further, if we think in the context of designing intelligent anti-malware systems, adversarial samples generated by augmenting large number of call sequences would deliberately force the application execute longer on the device. Thus, anti-viruses making use of simple heuristic such as the utilization of memory (virtual memory, cursor, dalvik ), CPU usage, number of processes created, etc, would identify such applications.

Poisoning attack using static features can be easily simulated, but considerable efforts are needed for injecting dynamic features. Especially in all cases, we observed that Random forest and non-linear classifiers such as DNN and 1D-CNN are difficult to be attacked. Besides CNN shows a good detection rate in identifying modified malicious samples and obfuscated samples, as its convolution operation is capable of identifying repeated patterns in different regions of files, be it a chunk of system call sequence or byte stream. Another important observation emerged from our experiments is that the knowledge of the feature set plays a very significant role in creating adversarial samples. Randomly

selecting attributes and injecting them into applications does not create a successful attack.

An attack can be practically demonstrated by modifying the decompiled source of a malicious app. Top-weighted features comprising permissions, APIs and app components can be inserted into the decompiled code. By progressively adding features in the AndroidManifest.xml and rebuilding it, and later resigning the app creates a modified version with extraneous attributes. In our approach feature addition is considered for maintaining functionality of the application. Although, in the case of APIs, we can shield the call to specific API by substituting the characters by applying mono-alphabetic substitution (identical to additive cipher). Here our implication is to replace a character with a new character based on the specific substitution key. This will generate an encoded representation of the API. Logically, creating a modified version of encoded API in this way resembles the creation of an obfuscated application. To maintain the functionality a decoder module can be plugged in the app, which regenerates the API call name at runtime. Further the original API is invoked through Java reflection. However, an evasion attack created by the above-mentioned strategy using API modification would fail while performing dynamic analysis, as the classifier designed on dynamic attributes can identify the call to decoded APIs during runtime. We left the implementation as an open research problem, which we plan to address in our future work.

Relying on the lessons learnt by conducting our experiments, in future we plan to propose countermeasures for evasion attack. Following are our proposal:

- Address  $N$  class problem as  $N + 1$  class problem. This means we must develop a proactive system wherein the designers of the anti-malware system must simulate the behaviour of an adversary. By doing this, a large collection of adversarial samples can be approximated. A set of created samples can be used to augment the training set. In other words, classifiers are trained using malware, benign and adversarial examples.
- Development of ensembles of classifiers randomly trained on subset of attributes that periodically are modified during the re-training process. As the knowledge of features is critical for crafting attacks, it will hinder attack tactics as an adversary is unaware of classifier revision and

**Table 13** Resume of the Related Work

Paper	Contributions
Patel and Buddadev [33]	Hybrid Android malware detection Permissions and behaviour-based features Rule generation
Wang et al. [46]	Hybrid malware detector Detection of zero days
Damodaran et al. [16]	Comparative analysis on malware detection system Static, dynamic, and hybrid analysis
Wu and Hung [47]	Static and dynamic features
Saracino et al. [38]	Experiment on KNN classifier
Li et al. [27]	Malware detection by mining permission SVM and decision trees for classification
Chuang and Wang [15]	Classification with frequency of API calls
Burguera et al. [11]	Dynamic analysis of Android apps Two means clustering algorithm
Dimjašević et al. [18]	Detection of Android malware through system calls
Afonso et al. [2]	Detection of Android malware API calls and system call traces
Garcia et al. [21]	Detection of Android malware Categorized Android API usage, reflection-based features, and Features from native binaries of apps
Tam et al. [43]	Reconstructing behaviors of Android malware Observing system calls
Almin and Chatterjee [4]	Analysis of permissions Clustering and classification techniques
Kim et al. [25]	Android malware detection Opcode features, API features, strings, permissions, app's Components, and environmental features
Sun and Qian [41]	Malware detection model-based on RNN and CNN
Ni et al. [32]	Opcode sequences, malware visualization, and deep learning
Saxe and Berlin [39]	Deep neural network Static features
Karbab et al. [24]	Deep learning techniques Raw sequences of API method calls
McLaughlin et al. [29]	Static analysis Raw opcode sequence from a disassembled program
Vinayakumar and Soman [45]	Comparison of deep neural networks(DNNs) andMachine learning algorithms for static malware detection
Le et al. [26]	Malware classification method using Visualization and deep learning
Agarap and Pepito [3]	Convolutional deep learning models
SI and CD [36]	CNN based windows malware detectorAPI calls
Martinelli et al. [28]	Convolutional neural network System calls
Xiao et al. [48]	Backpropagation neural network
Chen et al. [14]	Two-phase detection system
Xu et al. [49]	Genetic programming
Chen et al. [13]	Evading PDF malware classifiers
Grosse et al. [23]	Evaluation of standard classifiers
Chavan et al. [12]	Adversarial crafting attacks on neural network
Demontis et al. [17]	Experiments on permissions Binary and multiclass classification
Pierazzi et al. [35]	Adversary-aware machine learning detector
	Formalization of problem-space attacks Relationships between feature space and problem space

the features used to model the classifiers. Notably, the conclusion for assigning the labels for a sample under consideration could be based on *OR* operations, which means that if anyone among the pool of classifiers labels the sample as malware and all the others as legitimate, the target class label will be concluded as malware.

- Building classifier using a set of attributes that are difficult to be modified. This would restrict the attack surface as a modification to the aforementioned feature would affect the functionality of the program.

## 9 Conclusion and future work

In this paper, we present a study on malware detectors based on machine and deep learning classifiers, consisting of two experiments. In the first experiment, we propose a hybrid approach for malware detection, that lets us conclude that hybrid analysis increases the performance of classifiers concerning the independent features. The results show that with static features the SVM algorithm produces the best outcomes, and this corroborates the evidence provided by the literature. With regards to the dynamic analysis, the RF algo-

rithm showed better results, while the highest performances with the hybrid approach were obtained with CART and SVM algorithms. We extended our study by investigating the performances of the deep neural network, which also show that the hybrid features produced improved results.

In addition, we examined how evasion and poisoning attacks deteriorate the robustness of the classifiers. We showed that the evasion attack severely affects classifier performance with static features, however, evasive examples created using system calls (dynamic analysis) adversely affected the classifier outcome. We show a large collection of adversarial examples which are able to prevent from the detection. Concerning the classifiers, we observed that Random Forest and CNN offer a good resistance to adversarial attacks.

In the future, we will evaluate the performances of diverse deep learning models using multiple datasets. Additionally, we would like to test the reliability of classification systems on adversarial attacks trained on malware images techniques. In particular, we would like to explore how neurons in each layer participate in the feature extractor process.

## References

- Abou-Assaleh, T., Cercone, N., Keselj, V., Sweidan, R.: N-gram-based detection of new malicious code. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004., vol. 2, pp. 41–42. IEEE (2004)
- Afonso, V.M., de Amorim, M.F., Grégio, A.R.A., Junquera, G.B., de Geus, P.: Identifying Android malware using dynamically obtained features. *J. Comput. Virol. Hacking Techn.* **11**(1), 9–17 (2015)
- Agarap, A.F.: Towards building an intelligent anti-malware system: a deep learning approach using support vector machine (svm) for malware classification. arXiv preprint [arXiv:1801.00318](https://arxiv.org/abs/1801.00318) (2017)
- Almin, S.B., Chatterjee, M.: A novel approach to detect android malware. *Procedia Comput. Sci.* **45**, 407–417 (2015)
- Aonzo, S., Georgiu, G.C., Verderame, L., Merlo, A.: Obfuscapack: an open-source black-box obfuscation tool for Android apps. *SoftwareX* **11**, 100403 (2020)
- APKTool: <https://ibotpeaches.github.io/Apktool/install/>
- Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.E.R.T.: Drebin: effective and explainable detection of android malware in your pocket. In: Ndss, vol. **14**, pp. 23–26. (2014)
- Arshad, S., Shah, M.A., Wahid, A., Mehmood, A., Song, H., Hongnian, Y.: Samadroid: a novel 3-level hybrid malware detection model for android operating system. *IEEE Access* **6**, 4321–4339 (2018)
- Bernardi, M.L., Cimitile, M., Distanto, D., Martinelli, F., Mercaldo, F.: Dynamic malware detection and phylogeny analysis using process mining. *Int. J. Inf. Secur.* **18**(3), 257–284 (2019)
- Biggio, B., Fabio, R.: Wild patterns: ten years after the rise of adversarial machine learning. *Pattern Recognit.* **84**, 317–331 (2018)
- Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowddroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26. (2011)
- Chavan, N., Di Troia, F., Stamp, M.: A comparative analysis of android malware. arXiv preprint [arXiv:1904.00735](https://arxiv.org/abs/1904.00735) (2019)
- Chen, L., Hou, S., Ye, Y., Xu, S.: Droideye: fortifying security of learning-based classifier against adversarial android malware attacks. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 782–789. IEEE (2018)
- Chen, S., Xue, M., Fan, L., Hao, S., Xu, L., Zhu, H., Li, B.: Automated poisoning attacks and defenses in malware detection systems: an adversarial machine learning approach. *Comput. Secur.* **73**, 326–344 (2018)
- Chuang, H.Y., Wang, S.D.: Machine learning based hybrid behavior models for Android malware analysis. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 201–206. IEEE (2015)
- Damodaran, A., Di Troia, F., Visaggio, C.A., Austin, T.H., Stamp, M.: A comparison of static, dynamic, and hybrid analysis for malware detection. *J. Comput. Virol. Hacking Techn.* **13**(1), 1–12 (2017)
- Demonits, A., Melis, M., Biggio, B., Maiorca, D.A., Rieck, K., Corona, I., Giacinto, G., Roli, F.: Yes, machine learning can be more secure! a case study on android malware detection. In: IEEE Transactions on Dependable and Secure Computing, vol.16, pp. 711–723. IEEE (2019)
- Dimjašević, M., Atzeni, S., Ugrina, I., Rakamarić, Z.: Evaluation of android malware detection based on system calls. In: Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp. 1–8. (2016)
- Dong, Y., Liao, F., Pang, T., Su, H., Zhu, J., Hu, X., Li, J.: Boosting adversarial attacks with momentum. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 9185–9193. (2018)
- Gandotra, E., Bansal, D., Sofat, S.: Malware analysis and classification: a survey. *J. Inf. Secur.* **2014** (2014)
- Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **26**(3), 1–29 (2018)
- Greengard, S.: Cybersecurity gets smart. *Commun. ACM* **59**(5), 29–31 (2016)
- Grosse, K., Papernot, N., Manoharan, P., Backes, M.I., McDaniel, P.: Adversarial examples for malware detection. In: European Symposium on Research in Computer Security, pp. 62–79. Springer, Cham (2017)
- Karbab, E.B., Debbabi, M., Derhab, A., Mouheb, D.: MalDozer: automatic framework for android malware detection using deep learning. *Digit. Investig.* **24**, S48–S59 (2018)
- Kim, T.G., Kang, B.J., Rho, M., Sezer, S., Im, E.G.: A multimodal deep learning method for android malware detection using various features. *IEEE Trans. Inf. Forensics Secur.* **14**(3), 773–788 (2018)
- Le, Q., Boydell, O., Namee, B.M., Scanlon, M.: Deep learning at the shallow end: malware classification for non-domain experts. *Digit. Investig.* **26**, S118–S126 (2018)
- Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., Ye, H.: Significant permission identification for machine-learning-based android malware detection. *IEEE Trans. Ind. Inf.* **14**(7), 3216–3225 (2018)
- Martinelli, F., Marulli, F., Mercaldo, F.: Evaluating convolutional neural network for effective mobile malware detection. *Procedia Comput. Sci.* **112**, 2372–2381 (2017)
- McLaughlin, N., del Rincon, J.M., Kang, B.J., Yerima, S., Miller, S., Sakir, S., et al.: Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308. (2017)
- MonkeyRunner: <https://developer.android.com/studio/test/monkey>
- Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: visualization and automatic classification. In: Proceedings

- of the 8th International Symposium on Visualization for Cyber Security, pp. 1–7. (2011)
32. Ni, S., Qian, Q., Zhang, R.: Malware identification using visualization images and deep learning. *Comput. Secur.* **77**, 871–885 (2018)
  33. Patel, K., Buddadev, B.: Detection and mitigation of android malware through hybrid approach. In *International symposium on Security in Computing and Communication*, pp. 455–463. Springer, Cham, (2015)
  34. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: TESSERACT: eliminating experimental bias in malware classification across space and time. In: *28th USENIX Security Symposium (USENIX Security 19)*, pp. 729–746. (2019)
  35. Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L.: Intriguing properties of adversarial ML attacks in the problem space. In: *Proceedings of IEEE Symposium on Security and Privacy, 2020*, pp.1332–1349. IEEE (2020)
  36. SL, S.D., Jaidhar, C.D.: Windows malware detector using convolutional neural network based on visualization images. *IEEE Trans. Emerg. Top. Comput.* (2019)
  37. Santos, I., Penya, Y.K., Devesa, J., Bringas, P.G.: N-grams-based file signatures for malware detection. *ICEIS* **9**, 317–320 (2009)
  38. Saracino, A., Sgandurra, D., Dini, G., Martinelli, F.: Madam: effective and efficient behavior-based android malware detection and prevention. *IEEE Trans. Dependable Secure Comput.* **15**(1), 83–97 (2016)
  39. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20. IEEE (2015)
  40. Sen, S., Aydogan, E., Aysan, A.I.: Coevolution of mobile malware and anti-malware. *IEEE Trans. Inf. Forensics Secur.* **13**(10), 2563–2574 (2018)
  41. Sun, G., Qian, Q.: Deep learning and visualization for identifying malware families. *IEEE Trans. Dependable Secure Comput.* (2018)
  42. Surendran, R., Thomas, T., Emmanuel, S.: GSDroid: graph signal based compact feature representation for android malware detection. *Expert Syst. Appl.* **159**, 113581 (2020)
  43. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: automatic reconstruction of android malware behaviors. In: *Ndss*. (2015)
  44. Ucci, D., Leonardo, A., Roberto, B.: Survey of machine learning techniques for malware analysis. *Comput. Secur.* **81**, 123–147 (2019)
  45. Vinayakumar, R., Soman, K.P.: DeepMalNet: evaluating shallow and deep networks for static PE malware detection. *ICT Express* **4**(4), 255–258 (2018)
  46. Wang, X., Yang, Y., Zeng, Y., Tang, C., Shi, J., Xu, K.: A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In: *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services*, pp. 15–22. (2015)
  47. Wu, W.C., Hung, S.H.: DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, pp. 247–252. (2014)
  48. Xiao, X., Wang, Z., Li, Q., Xia, S., Jiang, Y.: Back-propagation neural network on Markov chains from system call sequences: a new approach for detecting Android malware with system call sequences. *IET Inf. Secur.* **11**(1), 8–15 (2017)
  49. Xu, W., Qi, Y., Evans, D.: Automatically evading classifiers. In: *Proceedings of the 2016 Network and Distributed Systems Symposium*, vol. 10. (2016)
  50. Xue, Y., Meng, G., Liu, Y., Tan, T.H., Chen, H., Sun, J., Zhang, J.: Auditing anti-malware tools by evolving android malware and dynamic loading technique. *IEEE Trans. Inf. Forensics Secur.* **12**(7), 1529–1544 (2017)
  51. Zhang, S., Xiao, X.: Cscdroid: Accurately detect android malware via contribution-level-based system call categorization. In *2017 IEEE Trustcom/BigDataSE/ICSS*, pp. 193–200. IEEE (2017)
  52. Zhou, M.: A hybrid feature selection method based on fisher score and genetic algorithm. *J. Math. Sci. Adv. Appl.* **37**(1), 51–78 (2016)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.