*Journal of Cyber Security*
DOI:10.32604/jcs.2021.016632
***Article***

Tech Science Press

# An LSTM-Based Malware Detection Using Transfer Learning

**Zhangjie Fu[1,2,3,*], Yongjie Ding[1] and Musaazi Godfrey[1]**

[1]School of Computer and Software, Nanjing University of Information Science and Technology, Nanjing, 210044, China
[2]Guangxi Key Laboratory of Cryptography and Information Security, Guilin, 541004, China
[3]College of Information Science and Technology, College of Cyber Security, Jinan University, Guangzhou, 510632, China
[*]Corresponding Author: Zhangjie Fu. Email: fzj@nuist.edu.cn
Received: 08 January 2021; Accepted: 11 January 2021

**Abstract:** Mobile malware occupies a considerable proportion of cyberattacks. With the update of mobile device operating systems and the development of software technology, more and more new malware keep appearing. The emergence of new malware makes the identification accuracy of existing methods lower and lower. There is an urgent need for more effective malware detection models. In this paper, we propose a new approach to mobile malware detection that is able to detect newly-emerged malware instances. Firstly, we build and train the LSTM-based model on original benign and malware samples investigated by both static and dynamic analysis techniques. Then, we build a generative adversarial network to generate augmented examples, which can emulate the characteristics of newly-emerged malware. At last, we use the augmented examples to retrain the 4th and 5th layers of the LSTM network and the last fully connected layer so that it can discriminate against newly-emerged malware. Actual experiments show that our malware detection achieved a classification accuracy of 99.94% when tested on augmented samples and 86.5% with the samples of newly-emerged malware on real data.

**Keywords:** Malware detection; long short term memory networks; generative adversarial networks; transfer learning; augmented examples

## 1 Introduction

The mobility and ever-evolving exceptional and fascinating features of mobile devices, e.g., smartphones have made us store confidential data and information—from personal information, corporate information, to other sensitive data into them. With sophisticated mobile malware evolving now and then, all the information kept on mobile devices is a target for this malware threat. Therefore, there is a need to continuously improve existing defense mechanisms against malware attacks, and as well innovate state of the art automated malware detection systems to secure our confidential information stored on mobile devices. There is an exponential growth in mobile malware complexity and scope every year, reaching 91778 in 2019 [1]. According to McAfee Labs Mobile Threat Report as of Q1 2020 [2], mobile malware has expanded the ways of hiding their attacks and frauds, making them increasingly difficult to identify and remove.

Mobile malware has been analyzed by researchers using static methods, dynamic methods, or a combination of both (hybrid analysis). In static analysis: features are extracted from the manifest file or the Java bytecode, without executing the code, while for dynamic analysis: extraction of features is done during code execution (or emulation). The hybrid analysis is a combination of both static analysis and dynamic analysis methods. It is meant to analyze malware from various aspects and is considered a better approach than standalone analysis methods.

Machine learning-based malware detection systems have been proposed in recent works, and they have provided promising results. However, their defense against constantly newly-emerged attacks [3] still poses a great challenge to researchers. Adversarial attacks [4,5] aim at misleading the classifier such that malicious apps are misclassified as benign (integrity attack) or creating a denial of service in which benign apps are incorrectly classified as malicious (availability attack). Therefore, there is a need for resilient detection systems that can defend against constantly newly-emerged attacks. Deep learning techniques are also being employed by researchers in the fight against malicious activities. Deep learning is a section of machine learning in the field of Artificial Intelligence(AI) [6]. Deep learning is defined as neural networks with a large number of parameters and layers in a given deep learning architecture. A deep neural network's process of deciding which characteristics of the dataset can be used as indicators to label input data reliably (Automatic feature extraction) is one of the great advantages of deep learning over traditional machine learning algorithms [7]. Deep learning has been applied and has demonstrated better performance than traditional machine learning algorithms [8,9].

In this research work, we present a new approach to building a robust mobile malware detection model that is able to detect constantly newly-emerged malware instances. The model is based on two Deep Learning architectures; Artificial Neural Network [10–12] and Generative Adversarial Network (GAN) [13]. The model's performance is boosted by retraining it with augmented examples crafted by a GAN. We build the model in phases; initially, we build, train, and test the model on original benign and malware samples investigated by both static [14,15] and dynamic [16,17] analysis techniques. Then we build a generative adversarial network and generate augmented examples. The model is tested on augmented samples before and additional samples after retraining with augmented examples.

The main contributions of our research work can be summarized as follows:

(1) We proposed a malware detection model based on LSTM to detect constantly newly-emerged malware instances. Our method not only achieves high accuracy for existing malware but also works for newly-emerged malware.

(2) For the first time, we build a malware detection modele using transfer learning to retrain, so it can fight against newly generated malware with undefined characteristics.

(3) The accuracy of our method has a good improvement compared to the previous method, which achieved 99.94% accuracy in malware detection on existing data sets and 86.5% accuracy on real data sets of new malware.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces our proposed methods and framework in detail. In Section 4 we present our findings and discussion of results. Lastly, we conclude the paper and also give highlights of future work in Section 5.

## 2 Related Works

### *2.1 Malware Analysis*

Cui et al. [18] presented a data mining approach through dynamic analysis of application behavior for detecting malware in the Android platform. Feature extraction (System calls) was done on the device while analysis and detection utilized cloud computing. The main challenge is that the approach can only distinguish between benign and malicious apps of the same name and version.

Martinelli et al. [19] presented a malware detection tool “BRIDESMAID” that utilized hybrid analysis techniques to detect Android malware. The authors' static analysis was based on n-grams matching, while the dynamic analysis was based on multi-level monitoring of device, app, and user behavior. They reported 99.7% detection accuracy on testing with 2794 malicious apps. However, constantly newly-emerged cases were not considered.

Zaki et al. [20] investigated the behavior of Mobile malware through a hybrid approach. The authors proposed a general mobile malware behavior model that can contribute in identifying the key features to detecting mobile malware on an Android platform device.

### *2.2 Machine Learning in Mobile Malware Detection*

Machine learning has been incorporated in most of the current research works. However, due to the sophistication of new unknown malware releases almost daily, most detection systems' performance is greatly affected when they fail to detect such malware, and this exposes mobile users to more deadly malware attacks.

In [21], a machine learning approach with Rotation forest classification was proposed. Extraction of permissions, sensitive APIs, monitoring system events, and permission-rate as key features, than employing the ensemble Rotation Forest (RF) to construct a model to detect whether an Android application is malicious or not was done. DroidDet was based on the static analysis methodology.

Ahmadi et al. [22] presented IntelliAV: an on-device tool that used a machine learning model built on top of the Tensor flow. APIs, Requested Permissions, and INT: Intents, CG was the features extracted. IntelliAV is based on a static analysis approach whose limitations include dynamic code loading techniques, and victim of evasion techniques against the learning approach.

### *2.3 Deep Learning in Mobile Malware Detection*

Su et al. [8] and Li et al. [9] presented malware detection approaches for the Android platform based on deep learning. Both used static analysis techniques and reported over 97% detection accuracy, but constantly newly-emerged attacks were not considered.

Yuan et al. [23] presented a deep learning-based malware detection model “DroidDetector” that could achieve 96.76% detection accuracy, which outperforms traditional machine learning techniques. Basing on their results, they noted that deep learning is suitable for characterizing Android malware and especially effective with the availability of more training data.

Makandar et al. [24] analyzed and classified malware using an algorithm they implemented using feed-forward Artificial Neural Networks (ANN). They reported a classification accuracy of 96.35%.

Several deep learning architectures including convolutional neural network (CNN), Artificial Neural Network (ANN), GAN, and Deep Brief Network (DBN) are utilized by different researchers to build more resilient detection models. For example, [25] proposed a novel Android malware detection system that used a deep convolutional neural network to process the raw Dalvik bytecode of an Android application.

Although the detection accuracies reported by several authors are promising, constantly newly-emerged attack scenarios have not been addressed. These methods have limited recognition of emerging malware that is increasingly dangerous.

### *2.4 Generated Mobile Malware Samples*

In previous works [26–28] generated malware examples have been used to mislead detection models into classifying malware apps as benign. Additionally, in [27,28] an investigation on potential defense mechanisms for hardening malware detection models trained using Deep Neural Networks (DNN) was conducted. Adversarial training of the model, intentionally with generated crafted malware applications was found to improve the model’s robustness, as long as the perturbation introduced during adversarial training is carefully chosen.

Wang et al. [29] assessed the transferability of adversarial examples generated on a sparse and structured dataset, and the ability of adversarial training of malware detection classifiers in resisting adversarial examples. Authors reported that adversarial examples generated by DNN can fool several machine learning classifiers–decision tree, random forest, SVM, CNN, and RNN. They further note that adversarial training can improve the robustness of DNN in terms of resisting adversarial attacks.

In [26], a GAN was used to craft generated malware examples for PC malware basing on 160 API features to fool a black box malware detector. A GAN comprises a generator network that takes random input and generates a sample of data and a discriminator network that takes input from the generator and

tries to predict whether the input is real or fake. The generator is trained to mislead the discriminator into misclassifying generated samples as real samples. In our work, we generate examples of Android malware using dynamic features.

## 3 Methodology

### *3.1 Overview*

In our proposed approach, we build and train a powerful model (SmartAMD2, Smart Android Malware Detection 2) in 5 stages. At first, we build, train, and test the model (SmartAMD1, Smart Android Malware Detection 1) on the original dataset (benign apps and malware apps), then we build a GAN and generate augmented examples. Hereafter, we test SmartAMD1 on augmented samples to verify its effectiveness. To transform SmartAMD1 into a robust model- SmartAMD2, we retrain its 4th and 5th layers LSTM network and the last layer is a fully-connected layer on augmented examples. This model will greatly improve the model's robustness incorrectly classifying malware including constantly newly-emerged malware examples. In the final stage, we test SmartAMD2 on additional examples. Fig. 1 shows our proposed approach.

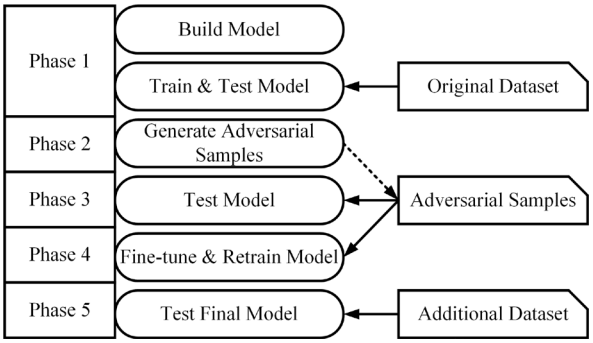


**Figure 1:** Methodological framework

### *3.2 Dataset*

We generate our dataset by analyzing extracting significant features of benign and malware apps for malware detection. The motivation to generate our dataset is because to cope with current Android security challenges. It is important to analyze up-to-date apps, and each research has its unique objectives. To diversify our research, we experiment with 6,180 samples from different but credible sources as described below.

#### *3.2.1 Benign Samples*

The benign .apk files were collected randomly from two sources: (1) ApkPure APP store: which is a source for the original free APK files in Google Play store which is considered as the official market with the least possibility of malware applications. (2) Android Wake Lock Research Project: which is a source for binaries (APK files) of commercial Android apps that use wake locks. We randomly collected 3090 benign samples in total.

**Table 1:** Benign APK samples

| Source | Number of Samples |
|---|---|
| Apkpure.com | 2090 |
| Android Wake Lock Research Project | 1000 |
| Total | 3090 |

*3.2.2 Malware Samples*

Malicious .apk files were collected from 2 sources: (1) Virusshare: which is a repository of malware samples, it provides security researchers, incident responders and forensic analysts access to samples of live malicious code; (2) Ashishb: it is a GitHub Android malware repository. We collected 3090 Android malware samples from the 2 sources.

**Table 2:** Malware APK samples

| Source | Number of Samples |
|---|---|
| Virusshare.com | 3019 |
| Android-malware (git repo) | 71 |
| Total | 3090 |

### *3.3 The Architecture of the Proposed Scheme*

We show the architectural design of our proposed approach of building a powerful malware detection that is retrained on augmented examples in Fig. 2. Sections 3.4, 3.5, 3.6 and 3.7 detail each of the steps shown in the proposed scheme.

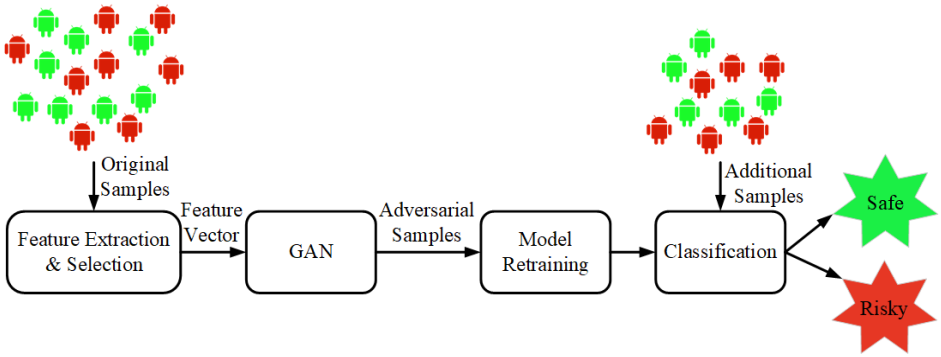


**Figure 2:** Architecture of the proposed scheme

### *3.4 Feature Extraction and Selection*

Hybrid features are the most comprehensive features because they analyze applications from various aspects. In this research work, therefore we utilize both static and dynamic features that are significant in malware detection.

*3.4.1 Static Features*

**Permissions:** Before installation of an Android application, the Android permission system controls the access to privacy and security-relevant APIs. According to the functionality of an application, it must declare which permissions it will use in the AndroidManifest.xml.

**Receivers and receivers' actions:** A receiver also known as a broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low or a picture was captured. Mohsen et al. [30] reported that Android Broadcast receivers are intensively used by malware compared to benign application.

*3.4.2 Dynamic Features*

Dynamic behaviors exhibited by apps during execution provide clues about malicious activities performed by suspicious apps. Monitoring and analyzing an app's behavior while in execution can aid in detecting malware. In experiments, we consider system calls, registered broadcast receivers at runtime,

and fingerprints for dynamic features. Every application demands resources and services from the operating system by issuing system calls, such as read, write, and open.

*3.4.3 Feature Extraction Methods*

To automate the feature extraction process, we make use of CuckooDroid by Check Point Software Technologies.

CuckooDroid is an automated, cross-platform, emulation and analysis framework based on the popular Cuckoo sandbox and several other open-source projects – providing both static and dynamic APK inspection, as well as evading certain VM-detection techniques, encryption key extraction, SSL inspection, API call trace, basic behavioral signatures and many other features [31].

Fig.3 below shows the process of analyzing each APK sample and how features are extracted.

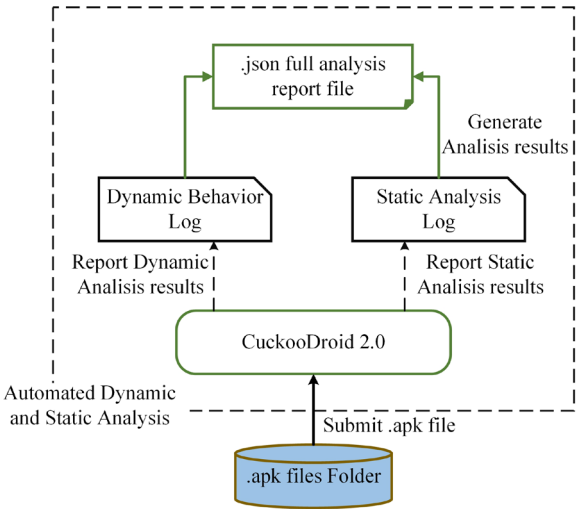


**Figure 3:** Analysis and feature extraction for an APK file

The APK analysis and features extraction process starts by submitting the APK files to CuckooDroid and then starting the analysis and feature extraction process described in Fig. 3. During the analysis, the Android emulator provided by Android Studio is started and Android Debug Bridge (ADB) installs the APK on the emulator where dynamic execution is monitored. Upon analysis completion, a JSON report with both static and dynamic analysis results is generated. We organize the APK files into 2 folders (malware folder and benign folder). The analysis is done firstly on benign files and their JSON reports stored in a folder, then analysis of malware APKs is also carried out and corresponding json reports are also stored in a folder. Before we can consider the analysis of a given sample as successful and therefore store its JSON report, we look at its analysis log and check if the sample (APK) installed successfully and was executed on the emulator.

*3.4.4 Feature Selection Method*

Determining which features are significant: in other words, features that greatly impact the outcome is one of the fundamental steps in building a powerful, effective, and efficient machine learning model. The more the number of features, the more computational power and time required for a model to run and produce results. In the beginning, we had 714 features (from both static and dynamic analysis) but after conducting feature selection on our data, we were able to consider only 357 most important features for our study. The objective of feature selection is three-fold: improving the prediction performance of the predictors, providing faster and more cost-effective predictors, and providing a better understanding of the underlying process that generated the data [32].

We use the random forest algorithm's method to determine the score for each of the 714 features.

Feature selection using Random forest comes under the category of embedded methods. Embedded methods combine the qualities of filter and wrapper methods. They are implemented by algorithms that have their built-in feature selection methods. Some of the benefits of embedded methods are high accuracy, better generalization, and interpretability.

The threshold for selecting features was set to $4\times10^{-6}$ and any feature that scored above the threshold was considered as a significant feature. Thus getting 357 significant features in total. Tab. 3 shows 20/357 most significant features we considered.

**Table 3:** Most significant features

| Features | Weights |
|---|---|
| android_util_Base64_encode | 0.074646 |
| android.permission.READ_PHONE_STATE | 0.064252 |
| android.permission.SEND_SMS | 0.060294 |
| android.intent.action.BATTERY_CHANGED | 0.040793 |
| getDeviceId | 0.035959 |
| dalvik_system_DexFile_dalvik_system_DexFile | 0.023681 |
| android.intent.action.BATTERY_CHANGED | 0.023639 |
| getSubscriberId | 0.021915 |
| android_content_ContentValues_put | 0.021404 |
| android_util_Base64_encodeToString | 0.020104 |
| dalvik_system_DexFile_loadDex | 0.018581 |
| android.permission.WAKE_LOCK | 0.017911 |
| android.permission.ACCESS_WIFI_STATE | 0.016962 |
| android.permission.GET_TASKS | 0.016792 |
| android_telephony_TelephonyManager_getSubscriberId | 0.013778 |
| java_net_ProxySelectorImpl_select | 0.013303 |
| android.permission.MOUNT_UNMOUNT_FILESYSTEMS | 0.013049 |
| android.permission.GET_ACCOUNTS | 0.012937 |
| android_telephony_TelephonyManager_getDeviceId | 0.012892 |
| android_app_SharedPreferencesImpl_EditorImpl_putLong | 0.011981 |

The JSON report contains a lot of information and we use a python script to extract only the data required for our study see Fig. 4. From the original report, we extract static and dynamic features. For static, we get the requested permissions, and receiver actions registered in manifest. Permissions like *android.permission.PROCESS-OUTGOING-CALLS* which allows the application to process outgoing calls and change the number to be dialed. Malicious applications may monitor, redirect, or prevent outgoing calls, *android.permission.READ-CONTACTS* this allows an application to read all of the contacts (address) data stored on your phone. Malicious applications can use this to send your data to other people, *android.permission.READ-PHONE-STATE*, which allows the application to access the phone features of the device. An application with this permission can determine the phone number and the serial number of the phone, whether a call is active, the number that calls are connected to, and so on. In total, we investigated 136 permissions. Lastly, we consider 74 receivers' actions, such as *android.intent.action.BOOT-COMPLETED*, *android.intent.action.DOWNLOAD-COMPLETE*, etc.

Therefore, for static analysis, we gather 210 features. The script accesses the folder containing all JSON files and loops over them, for each report, it gets the md5 of the APK that was analyzed together

with the data we need plus classification value (0 for benign and 1 for malware) and appends it to final JSON that stores all data about both malware and benign APK samples and this forms our dataset in JSON format.

For dynamic behavior, we looked for Registered Receivers during Runtime such as: *android.intent.action. PROXY-CHANGE*, *android.intent.action.SCREEN-OFF*, *android.intent.action.USER-PRESENT* and in total 64 registered receivers were considered. We record 72 API calls including but not limited to: *android-app-ContextImpl-registerReceiver*, *java-io-File-exists*, and *libcore-io-IoBridge-open*. Also a total of 11 fingerprints like *getSimOperatorName*, *getSubscriberId* and *getDeviceId* were investigated. Therefore, we collected 147 dynamic behaviors. This makes a grand total of 357 features to support our investigation in this research work.

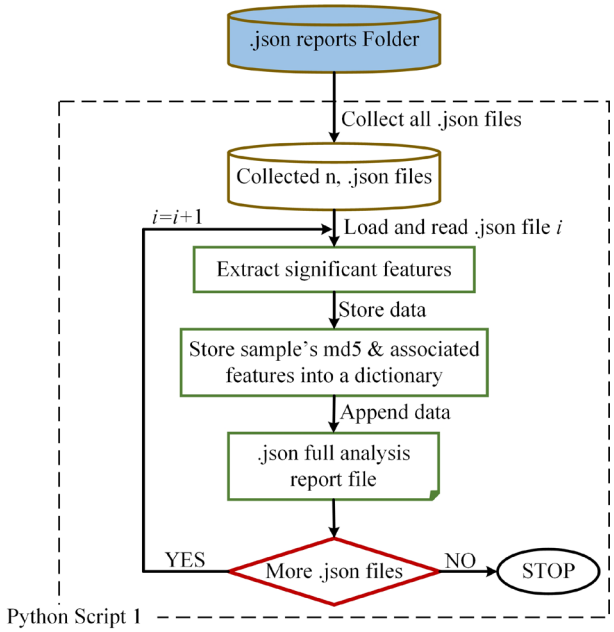


**Figure 4:** Generating Dataset for All Samples

### *3.5 Our Proposed Network Model of SmartAMD1*

#### *3.5.1 Overview*

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture [33] used in the field of deep learning. It can avoid the problem of gradient disappearance of RNN and has a stronger memory ability than ordinary RNN. Therefore, for the same time series, events with relatively long intervals or delays can also be efficiently processed and predicted. It should be noted that this excellent ability of the LSTM network is not obtained by algorithm learning, but by the inherent advantages of its unit structure. LSTM can determine the forgetting or retention of information through the memory controller, and then complete the input and output of information through the forget gate, input gate, and output gate.

Different types of artificial neural network topographies are suited for solving different types of problems. After determining the type of given problem we need to decide on the topology of the artificial neural network we are going to use and then fine-tune it. We need to fine-tune the topology itself and its parameters [10]. We need to fine-tune the topology itself and its parameters [10].

#### *3.5.2 Our model of SmartAMD1*

SmartAMD1 shown in Fig. 5 is a multilayer perceptron based feed-forward fully connected neural network. The network consists of 8 layers, with 3 LSTM layers and 3 fully connected layers, and the

number of neurons from input to output layers are as follows: L0:357, L1:180, L2:90, L3:30, L4:12, and L5:8. Batch size is set to 100, and maximum epochs are 800. We use Adam Optimizer, Rectified linear units (ReLU) as the activation function for hidden layers, and sigmoid for the output layer. Implementation is done in Keras on Tensor-Flow Backend.

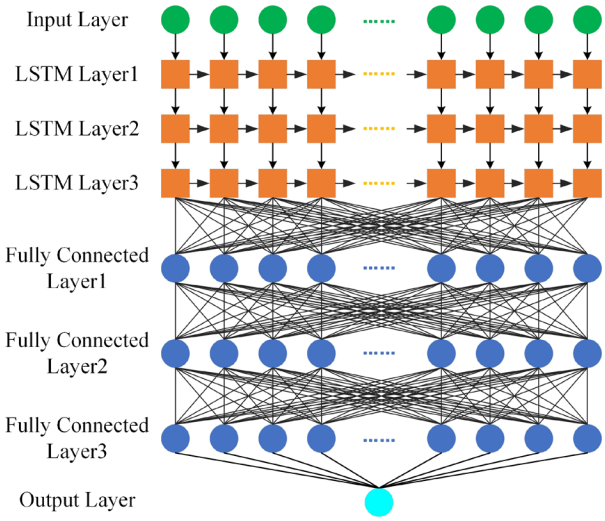


**Figure 5:** Architecture of SmartAMD1

### *3.6 Our Proposed Network Model of the GAN*

#### *3.6.1 Overview*

A GAN is composed of a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G [13]. GAN was used in [26], to transform original samples into augmented samples. Authors used binary features (160 APIs) to generate adversarial computer examples that could bypass machine learning black-box detectors.

In our case, we generate augmented examples for Android malware using dynamic features; the architecture used is based on that proposed by [26]. Generated augmented samples are used for additional training of SmartAMD1 thus creating a more robust model SmartAMD2. The Generator and Discriminator are both multi-layer feed-forward neural networks that work together to fool SmartAMD1. The feedback from the black box detector is used while generating augmented samples.

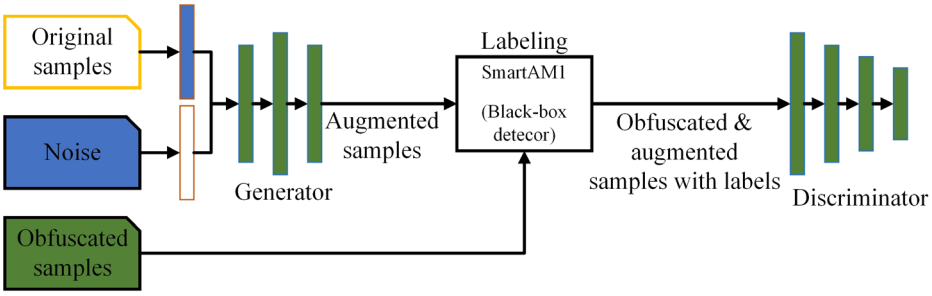


**Figure 6:** Architecture of GAN

#### *3.6.2 Generator*

The Generator intelligently adds certain irrelevant features to a malware feature vector to transform it into its augmented version. The concatenation of a noise vector $v$ and malware feature vector $m$ acts as input for the generator. $m$ is a binary m-dimensional feature vector, where the absence of a feature is

represented by 0 and its presence is represented by 1. A hyper-parameter $v$ allows the generator to generate diverse augmented examples from original feature vectors. $v$ is a v-dimensional vector whose elements are random numbers sampled from a uniform distribution in range (0–1).

The generator receives the input vector with weights $\theta_g$. Its output layer has m-neurons and uses a sigmoid activation function which limits the output in range (0–1). The generator's output is represented as $a$. Because malware feature values are binary, we apply binarization transformation to $a$ by assigning 1 if a feature value is greater than 0.5, otherwise, we assign 0. This results in the production of $a'$.

Generating augmented examples for binary features only requires adding certain irrelevant features to malware. The original malware may crack if a certain feature is removed. For example, an application that performs normal writing function will crack if the *"WRITE-EXTERNAL-STORAGE"* permission is removed. The irrelevant features to be added to the original malware are the non-zero elements of the binary vector $a'$. The final generated augmented example can be stated as $m' = m \mid a'$ where "|" is element-wise binary OR operation.

$m'$ is a binary vector, which implies that gradients cannot backpropagate from the discriminator to the generator. A smooth function G is defined to receive gradient information from the discriminator, as shown in (1).

$$G_{\theta_g}(m, v) = max(m, a) \tag{1}$$

$max(..)$ denotes element-wise max operation. If an element of $m'$ has the value 1, the corresponding result of G is also 1, which is unable to backpropagate the gradients. If an element of m has the value 0, the result of G is the neural network's real number output in the corresponding dimension, and gradient information can go through. It can be seen that $m'$ is actually the binary transformed version of $G_{\theta_g}(m, v)$.

### *3.6.3 Discriminator*

The discriminator is a multi-layer feed-forward neural network with weights $\theta_d$ and its input is a sample feature vector $x$. Its task is to classify the sample as a benign application or malware. The probability that $x$ is malware is represented as $D_{\theta_d}(x)$. The discriminator is trained on benign samples and augmented samples from the generator. The discriminator's goal is to fit the black-box detector and offer gradient information to train the generator. The ground-truth labels of the training data are not used to train the discriminator. The black-box detector will detect this training data first and output whether a sample is benign or malware. The predicted labels from the black-box detector are used by the discriminator.

### *3.6.4 Training GAN*

The loss function of the discriminator is defined in (2).

$$L_D = -E_{x \in BB_{Benign}} Log(1 - D_{\theta_d}(x)) - E_{x \in BB_{Malware}} logD_{\theta_d}(x) \tag{2}$$

$BB_{Benign}$ is the set of applications that are recognized as benign by the black-box detector, and $BB_{Malware}$ is the set of applications detected by the black box as malware.

To train the discriminator, $L_D$ should be minimized concerning the weights of the discriminator.

The loss function of the generator is defined in (3).

$$L_G = E_{m \in S_{Malware}, v \sim p_{uniform(0,1)}} logD_{\theta_d}(G_{\theta_g}(m, v)) \tag{3}$$

$S_{Malware}$ is the actual malware dataset, not the malware set labeled by the black-box detector. $L_G$ is minimized concerning the weights of the generator. Minimizing $L_G$ will reduce the predicted malicious probability of malware and push the discriminator to recognize malware as benign. Since the discriminator tries to fit the black-box detector, the training of the generator will further fool the black-box detector. The whole training process is shown in pseudo-code below:

**Algroithm 1** Training process

1: **while** not converging do
2: sample a mini batch of samples $m$
3: generate augmented samples $m'$ from the generator
4: sample a mini batch of obfuscated app $B$
5: label $m'$ and $B$ using a black box detector
6: update discriminator's weights $\theta_d$ by descending along the gradient $\nabla_{\theta_d} L_D$
7: update generator's weights $\theta_g$ by descending along the gradient $\nabla_{\theta_g} L_G$
8: **end while**

In Steps 2 and 4, different sizes of mini-batches are used for malware and benign applications. The ratio of $m'$'s size to $B$'s size is the same as the ratio of the malware dataset's size to the benign dataset's size.

### *3.7 Our Proposed Network Model of SmartAMD2*

#### *3.7.1 Overview*

Transfer learning, as the name suggests, is to transfer the learned model parameters to the new model to help the new model training [34]. Transfer learning can help us to quickly adjust the network structure and weights trained by the existing data set to enable it to complete the identification of the new data set. Such adjustments only need to change a small part of the network structure, and the training can be completed quickly.

In order to make our model adapt to the constantly updated software environment, transfer learning is used to retrain our model (SmartAMD1) so that it can transfer to our new model (SmartAMD2). The augmented simples generated by GAN are used to retrain this model so that it can be equally effective against emerging malware.

#### *3.7.2 Our Model of SmartAMD2*

SmartAMD2 adjusts the structure of the network on SmartAMD1 and retrains it with augmented samples. SmartAMD2 shown in Figure 7 adds two layers of LSTM network compared to SmartAMD1. The model solidifies the weights trained in SmartAMD1 so that it can take effect on the existing malware. On this basis, SmartAMD2 adds LSTM layer4 and LSTM layer5 after the first three LSTM layers of SmartAMD1. In addition, we also retrain the last fully connected layer of SmartAMD1. Through the training of LSTM layer4, LSTM layer5, and fully connected layer3, SmartAMD2 can take effect on newly augmented malware.

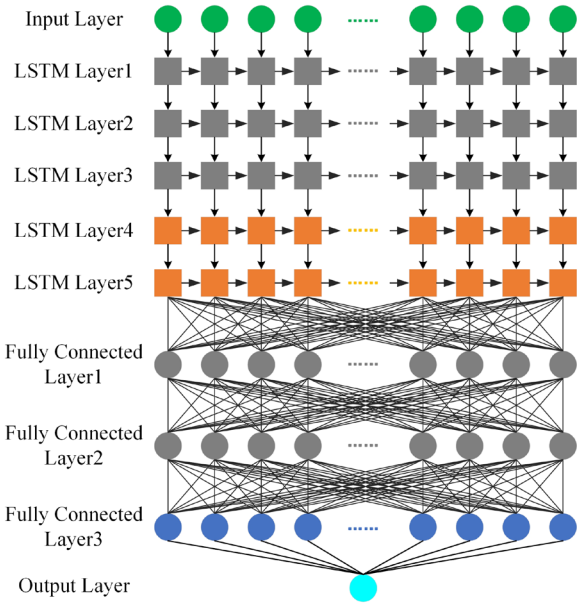


**Figure 7:** Architecture of SmartAMD2

## 4 Results and Discussion

### *4.1 Building, Training, and Testing SmartAMD1*

In building and training our deep learning model, we utilize two deep learning architectures; Long Short-Term Memory (LSTM) network, and a Generative Adversarial Network (GAN). For the deep learning framework; we use Keras running on top of Tensor Flow. In the next step, we train our model (SmartAMD1) based on deep learning techniques using extracted features from both benign and malicious apps. To conclude phase 1, we test SmartAMD1 on both benign and malicious apps.

#### *4.1.1 Data Preprocessing*

The model to be built, trained, and tested deals with binary data. Therefore at this stage, we import the JSON dataset and transform it into a pandas data frame by recording the presence of a particular feature as 0 for absent and 1 for a present for all samples identified by their md5 in the JSON dataset. Figure 8 shows the data preprocessing. This creates a two-dimensional array of bits.

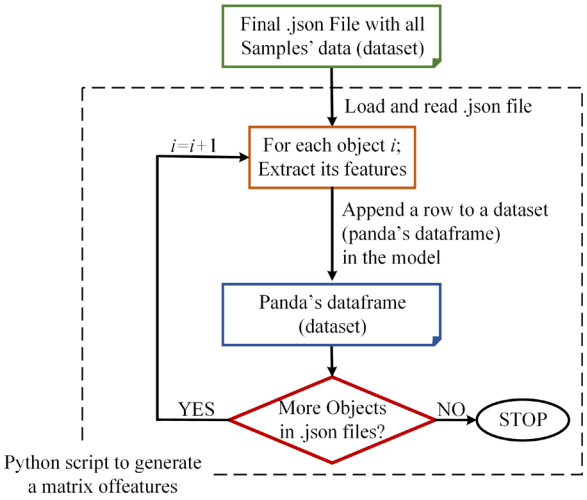


**Figure 8:** Generating a pandas matrix of features from JSON Dataset

*4.1.2 Training and testing SmartAMD1*

In this experiment, we train a multilayer feed-forward fully connected neural network (SmartAMD1) using the original dataset. To achieve the finest settings for optimal performance of our model, we vary the settings as reflected in Tab. 4 below. As you can note from the results table, the best accuracy of 99.88% was achieved with 357:180:90:30:12:8:1 layer size, 80 batch size, 800 epochs, and 7:3 splitting ratio for training and testing sets, respectively.

**Table 4:** Results (Original dataset)

| SmartAMD1 | Dataset | Samples | Layer size | Batch Size | Epochs | Accuracy |
|---|---|---|---|---|---|---|
| Version.1 | Training set | 4326 | 357:180:90:30:12:8:1 | 100 | 800 | 99.84% |
| | Testing set | 1854 | | | | |
| Version.2 | Training set | 4326 | 357:180:90:30:12:8:1 | 100 | 1000 | 99.87% |
| | Testing set | 1854 | | | | |
| Version.3 | Training set | 4326 | 357:180:90:30:12:8:1 | 80 | 800 | 99.88% |
| | Testing set | 1854 | | | | |

To show the model's weakness in detecting augmented samples, we test with a sample batch of 600 augmented examples. We see that all versions achieved excellent performance in detecting original samples but when tested on augmented samples, their True Positive Rates (TPRs) are reduced to almost 0 where SmartAMD1-Version.1 has 0.60% TPR, SmartAMD1-Version.2 has 0.80% TPR, and SmartAMD1-Version.3 has 0.30% TPR. Confusion matrices for each SmartAMD1 version are shown in Tab. 5 below, we can notice a high Type II error where almost all augmented malware is being misclassified as benign apps.

**Table 5:** SmartAMD1-Confusion matrix

| Version | | Malware | Benign |
|---|---|---|---|
| Version.1 | Malware | 4 | 596 |
| | Benign | 0 | 0 |
| Version.2 | Malware | 5 | 595 |
| | Benign | 0 | 0 |
| Version.3 | Malware | 2 | 598 |
| | Benign | 0 | 0 |

***4.2 Generating Augmented Examples***

The goal for this experiment is to generate augmented samples using the original dataset that was used in Section 4.1.2 above and to attack a black box detector using augmented samples.

We use the following configurations for the GAN. The noise vector has a dimension of 40, and feature dimensions are 357 which makes the combined input for the generator to be 397. Layer size for the generator is set to 397:700:357 whereas for discriminator it is set to 357:700:1. We use Binary Cross-Entropy for loss, Adam [35] as optimizer with a learning rate of 0.001, and accuracy as metrics. Generator and Discriminator networks both use sigmoid as activation function, which sets output values in the range (0, 1). We run 800 epochs with batch size 80.

We ran the experiment seven times and achieved the following results in Table 6. Augmented samples generated during each run were saved and later used as input for retraining SmartAMD1 into a more robust model SmartAMD2 and testing it as discussed in Section 4.3.

**Table 6:** Results of GAN experiments

| Experiments | Training set | | Test set | |
|---|---|---|---|---|
| | Original samples | Augmented samples | Original samples | Augmented samples |
| 1 | 97.32% | 0.65% | 97.66% | 0.65% |
| 2 | 97.36% | 0.35% | 97.54% | 0.39% |
| 3 | 97.62% | 0.17% | 96.76% | 0.52% |
| 4 | 97.45% | 0.39% | 97.28% | 0.39% |
| 5 | 97.24% | 0.65% | 97.93% | 0.39% |
| 6 | 97.32% | 0.43% | 97.67% | 0.52% |
| 7 | 97.37% | 0.39% | 97.54% | 0.26% |

From Fig. 9 (produced by experiment 5 in Tab. 6), we notice that the curve is not smooth because training a GAN network is unstable and is one of the open issues for research in GAN technology. From the $200^{th}$ epoch, we notice a TPR (True Positive Rate) convergence that becomes steady from $450^{th}$ epoch. Other experiments also showed similar graphs where convergence appears after $200^{th}$ epoch.

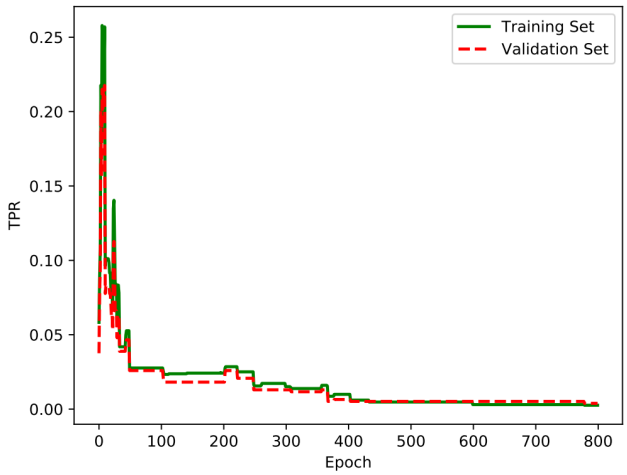


**Figure 9:** Variation of true positive rate

### *4.3 Boosting SmartAMD1 Hence SmartAMD2*

In this experiment we boost the performance of SmartAMD1, we retrain it on a dataset that just contains augmented samples such that it can learn the correlations among features of augmented samples, hence becoming a more robust model (SmartAMD2). We further test SmartAMD2 on additional samples to demonstrate its robustness. The architecture used for SmartAMD2 is that for the best model (Version.2) from Tab. 4. Experiments revealed that after retraining, SmartAMD2 can achieve 99.94% accuracy moreover with additional samples. We go ahead and test this SmartAMD2 on a batch of unseen additional samples and it could detect them with an accuracy of 86.5%. Tab. 7 and Table 8 below, show the results of the experiment.

**Table 7:** SmartAMD2 results

| Dataset | Samples | Layer size | Batch Size | Epochs | Accuracy |
|---|---|---|---|---|---|
| Training set | 4536 | 357:180:90:30:20:15:12:8:1 | 100 | 800 | 99.94% |
| Testing set | 1944 | | | | |
| Testing (additional samples) | 600 | | - | - | 86.50% |

**Table 8:** SmartAMD2-Confusion matrix

| | Malware | Benign |
|---|---|---|
| Malware | 513 | 87 |
| Benign | 0 | 0 |

We evaluate the performance of SmartAMD2 using the AUC (Area Under the Curve)—ROC (Receiver Operating Characteristics) curve. AUC is considered the most important evaluation metric for checking any classification model's performance. AUC is based on the output of a Confusion Matrix. To visualize the performance of SmartAMD2, we generate a ROC curve shown in Figure 10, we can see a higher AUC of 0.987 which implies that SmartAMD2 can distinguish well between benign and malware classes moreover including additional samples.

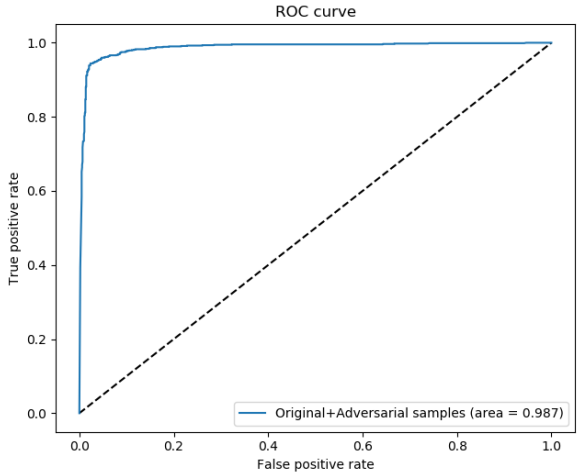


**Figure 10:** Variation of true positive rate

### *4.4 Comparison with other Methods*

In order to verify the effectiveness of our method, we conducted a series of comparative experiments with the methods proposed by others before us. The experiments showed that our method was more accurate than the previous methods and had an effect on the newly emerged malware.

Our model achieved a classification accuracy of 99.94% when tested on augmented samples and 86.5% with additional samples. Naive Bayes has the worst detection effect, with an accuracy rate of the only 80.6%; while the CNN with the best detection effect has an accuracy rate of 97.2%, and the accuracy of SmartAMD2 is 2.74% higher than it. The comparison with other methods is shown in Tab. 9.

**Table 9:** Comparison of Accuracy of Several Methods

| Method | Precious | Recall | F-measure | Accuracy(%) |
|---|---|---|---|---|
| KNN | 0.95 | 0.97 | 0.96 | 92.23 |
| SVM | 0.97 | 0.97 | 0.97 | 96.89 |
| RF | 0.99 | 0.98 | 0.98 | 98.32 |
| CNN-0 | 0.99 | 0.98 | 0.98 | 97.80 |
| CNN-S | 0.99 | 0.99 | 0.99 | 99.82 |
| DAE | 0.96 | 0.98 | 0.98 | 94.72 |
| Decision Tree | 0.98 | 0.98 | 0.98 | 97.54 |
| SmartAMD2 | **0.99** | **0.99** | **0.99** | **99.94** |

Compared with the CNN-S method proposed by Wang et al. [29], our method improves the recognition accuracy from 99.82% to 99.94%, far higher than KNN, SVM, RF, and other traditional methods. At the same time, the previous method cannot achieve good results in the detection of new malware, while our method can achieve an accuracy rate of 86.5% in the detection of new malware.

## 5 Conclusion

As new malware continues to appear, its harmfulness is also increasing. This calls for a greater need to build robust intelligent malware detection algorithms that can offer better protection in the arms race with malware authors.

The research presented in this paper proposes a framework to build a robust malware detection model that can detect malware including constantly newly-emerged malware examples. We have built a robust malware detection that we trained on benign samples, malware samples, and augmented samples using a combination of static and dynamic features. This malware detection can effectively combat newly emerged malware.

In the future, we intend to implement a cloud-based and distributed malware detection system and also more features will be considered. At the same time, we need to collect more and more malicious samples to train our model to make it more robust.

**Funding Statement:** Funding Statement: This work was supported by the National Nature Science Foundation of China (Nos. U1836110, 1836208).

**Conflicts of Interest:** The authors declare that they have no conflicts of interest to report regarding the present study.

## References

[1] V. Chebyshev. Mobile malware evolution 2019, 2019. [Online]. Available: https://securelist.com/mobile-malware-evolution-2019/96280.

[2] McAfee. McAfee Mobile Threat Report, 2020. [Online]. Available: https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf.

[3] Z. Abaid, M. A. Kaafar and S. Jha, "Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers," in *2017 IEEE 16th Int. Sym. on Network Computing and Applications (NCA)*. Cambridge, MA, UAS, pp. 1–10, 2017.

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

[5] N. Akhtar, and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14410–14430, 2018.

[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[7] J. Patterson and A. Gibson, "Deep learning: A practitioner's approach," in *O'Reilly Media*, Inc., Boston, MA, USA, 2017.

[8] X. Su, D. Zhang, W. J. Li and K. Zhao, "A deep learning approach to android malware feature learning and detection," in *2016 IEEE Trustcom/BigDataSE/ISPA*, Tianjin, China, pp. 244–251, 2016.

[9] D. F. Li, Z. G. Wang and Y. B. Xue, "Fine-grained android malware detection based on deep learning," in *2018 IEEE Conf. on Communications and Network Security (CNS)*, Beijing, China, pp. 1–2, 2018.

[10] A. Krenker, J. Bester and A. Kos, "Introduction to the artificial neural networks," in *Artificial Neural Networks-Methodological Advances and Biomedical Applications*, pp. 1–18, 2011.

[11] H. Alshahrani, A. Alzahrani, A. Alshehri, R. Alharthi and H. R. Fu, "Evaluation of gradient descent optimization: using android applications in neural networks," in *2017 Int. Conf. on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, pp. 1471–1476, 2017.

[12] N. A. Sammarraie, Y. M. H. Mayali and Y. A. B. Ebiary, "Classification and diagnosis using back propagation Artificial Neural Networks (ANN)," in *2018 Int. Conf. on Smart Computing and Electronic Enterprise (ICSCEE)*, Kuala Lumpur, Malaysia, pp. 1–5, 2018.

[13] I. Goodfellow, J. P. Abadie, M. Mirza, B. Xu, D. W. Farley, S. Ozair *et al.,* "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, Montreal, Quebec, Canada, pp. 2672–2680, 2014.

[14] S. Schmeelk, J. F. Yang and A. Aho, "Android malware static analysis techniques," in *Proc. of the 10th Annual Cyber and Information Security Research Conference*, Oak Ridge, TN, USA, pp. 1–8, 2015.

[15] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.

[16] A. Mahindru and P. Singh, "Dynamic permissions based android malware detection using machine learning techniques," in *Proc. of the 10th Innovations in Software Engineering Conf.*, Jaipur, India, pp. 202–210, 2017.

[17] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *Int. Conf. on Intelligent Communication and Computational Techniques (ICCT)*, Jaipur, India, pp. 1–7, 2017.

[18] S. J. Cui, G. X. Sun, S. Bin, and X. C. Zhou, "An android malware detection system based on cloud computing,' *Chemical Engineering Transactions*, vol. 51, pp. 691–696, 2016.

[19] M. Fabio, F. Mercaldo and A. Saracino, "Bridemaid: An hybrid tool for accurate detection of android malware," in *Proc. of the 2017 ACM on Asia Conference on Computer and Communications Security*, New York, NY, USA, pp. 899–901, 2017.

[20] M. M. Zaki, S. Sahib, M. F. Abdollah, S. R. Selamat, R. Yusof and R. Ahmad, "Profiling mobile malware behaviour through hybrid malware analysis approach," in *2013 9th Int. Conf. on Information Assurance and Security (IAS)*, Regensburg, Germany, pp. 78–84, 2013.

[21] H. J. Zhu, Z. H. You, Z. X. Zhu, W. L. Shi, X. Chen *et al.,* "DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, 2018.

[22] M. Ahmadi, A. Sotgiu and G. Giacinto, "Intelliav: Building an effective on-device android malware detector," arXiv preprint arXiv:1802.01185, 2018.

[23] Z. L. Yuan, Y. Q. Lu and Y. B. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[24] A. Makandar and A. Patrot, "Malware analysis and classification using artificial neural network," in *2015 Int, Conf, on Trends in Automation, Communications and Computing Technology (I-TACT-15)*, Bangalore, India, pp. 1–6. IEEE, 2015.

[25] N. McLaughlin, J. M. D. Rincon, B. Kang, S. Yerima, P. Miller *et al.,* "Deep android malware detection," in *Proc. of the Seventh ACM on Conference on Data and Application Security and Privacy*, Scottsdale, AZ, USA, pp. 301–308, 2017.

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

[27] K. Grosse, N. Papernot, P. Manoharan, M. Backes and P. McDaniel, "Adversarial examples for malware detection," in *European Sym. on Research in Computer Security*, Oslo, Norway, pp. 62–79, 2017.

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

[29] Y. X. Wang, J. Q. Liu and X. L. Chang, "Assessing transferability of adversarial examples against malware detection classifiers," in *Proc. of the 16th ACM Int. Conf. on Computing Frontiers*, New York, NY, USA. pp. 211–214, 2019.

[30] F. Mohsen, H. Bisgin, Z. Scott and K. Strait, "Detecting android malwares by mining statically registered broadcast receivers," in *2017 IEEE 3rd Int. Conf. on Collaboration and Internet Computing (CIC)*, San Jose, CA, USA, pp. 67–76, 2017.

[31] Check Point Software Technologies LTD. Global XMPP Android Ransomware Campaign Hits Tens of Thousands of Devices, 2015. [Online]. Available: https://blog.checkpoint.com/2015/08/31/global-xmpp-android-ransomware-campaign-hits-tens-of-thousands-of-devices/.

[32] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection." *Journal of Machine Learning Research*, vol. 3, no. 3, pp. 1157–1182, 2003.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] S. Ben-David, J. Blitzer, K. Crammer and F. Pereira, "Analysis of representations for domain adaptation," in *Advances in Neural Information Processing Systems*, Vancouver, BC, Canada, pp. 137–144, 2007.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv preprint arXiv:1412.6980, 2014.