EnsembleDroid: A Malware Detection Approach for Android System based on Ensemble Learning

Sharon Guan
College of Engineering and Applied Sciences
Stony Brook University
Stony Brook, NY 11794
sharon.guan@stonybrook.edu

Wenjia Li
Department of Computer Science
New York Institute of Technology
New York, NY 10023
wli20@nyit.edu

2022 IEEE MIT Undergraduate Research Technology Conference (URTC) | 978-1-6654-7345-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/URTC56832.2022.10002213

***Abstract*—In recent years, mobile devices such as smartphones and tablets have become one of the most popular digital devices of choice in our daily lives when it comes to functionality and convenience. With only ever increasing popularity, mobile devices with Android operating systems have become a common target for malware especially through third-party markets. To make things worse, the emergence of obfuscation and adversarial example attacks enables malware to evade traditional security methods and steal a user's private information. In this paper, we propose an ensemble learning- based framework for detecting malware by using risky permissions as features to train a classifier and determine whether a mobile application (a.k.a. app) is malware or not. To evaluate the performance of the proposed malware detection approach, we have conducted a series of experiments using real world Android app datasets that are composed of both malicious and benign apps. Experimental results clearly show that the proposed malware detection approach can effectively detect malware with a high accuracy.**

***Index Terms*—Android, security, malware detection, machine learning, ensemble learning**

I. Introduction

With such advanced technologies in the modern world, mobile devices including smartphones and tablet computers have been a popular choice of device for completing many tasks ranging from communication to entertainment. The most widely used operating system for mobile devices is Android, dominating about 87% of the global market share [1]. Given that the number of Android devices are predicted to grow even more, one would expect the mobile operating system to be secure with a user's personal data and information well kept against unwanted sharing and disclosure, which is the main goal of hackers when developing malicious applications (apps). However, that is far from the truth. In contrast, distribution of malicious Android apps, especially through third-party markets, have been gaining in numbers of up to 48,000 newly detected malware samples per month as of March 2020 [2].

While there have been many prior research works to address the security risks which are brought by various malicious apps (a.k.a. malware), their detection rates only range from 20.2% to 79.6% [3]. In addition, the Google Bouncer, which is the security solution deployed by the official Android market Google Play, has its own limitations. For instance, it can only scan Android apps for a limited time. A malicious app can simply surpass Google Bouncer by not doing anything during the scanning phase. Moreover, another limitation of Google Bouncer is that an app may contain no malicious code at the time of initial installation and make it seem safe to Google Bouncer. After it is installed, the application may download additional malicious code that can grant itself additional commands and controls from remote access. These additional commands and controls may allow the user's private and sensitive data to be shared without the user's own consent, which is a clear security breach. Other prior research works like DroidMat [4] and DroidAPIMiner [5] have their own limitations too. DroidMat is based on a K-Nearest Neighbor approach where its recall value is significantly lower than its precision value, which makes it unable to detect some types of malware. DroidAPIMiner's efficiency depends on a large amount of benign apps and it is also very time consuming.

In this paper, we propose EnsembleDroid, a malware detection approach for the Android system using the ensemble learning algorithm. Specifically, we have integrated both homogeneous stacking and heterogeneous stacking models when using the ensemble learning algorithm. Stacking allows us to produce several predictions, which serve as the input for a final meta classification model to distinguish malware from benign applications. By using the real world Android application datasets containing both malware and benign apps, we train both of the stacking ensemble learning classifiers, and the detection accuracy is 91% for both homogeneous and heterogeneous stacking models.

The rest of the paper is organized as follows. Section II will present the prior research works related to this research. Second III will focus on the methodology, and section IV will go through the experiment setup and discuss the results. Finally, Section V concludes this work and also presents some possible future directions.

II. Related Work

There have been many previous research efforts for detecting malware in the Android system. In this section, we will categorize them and summarize some research efforts for each category.

978-1-6654-7345-3/22/$31.00 ©2022 IEEE

A. Static Analysis

Static analysis is one of the widely used approaches for detection. It extracts static features such as permissions, API calls, URLs, and etc., thus being called static analysis. Then, static analysis searches for the similarities between previously existing malware and the targeted application. Kirin [8] mainly uses decompiling and data flow tracking as its main approach but requires manually-input malware patterns in order to properly detect the malware. Another example is mining permissions of both benign and malicious applications to be used to train a classifier that will distinguish the future permission patterns as malware [18]. The advantage of static analysis is that it scans for suspicious patterns and signatures without executing them and runs in a relatively short time compared to dynamic analysis. However, there are some limitations to static analysis as it cannot identify security vulnerabilities during run time.

B. Dynamic Analysis

Dynamic analysis works by analyzing for issues while an Android application is running in a safe environment. This may be done by running the application through a sandbox, an isolated testing environment that does not affect the platform or system that the program is in, or by installing it on real devices that can gather information about the run-time behavior. During that time, it monitors for suspicious activity that allows unwanted third parties to obtain sensitive and private data. TaintDroid [17] automatically labels data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. Any tainted data that is transmitted to the network or leaves the system is labeled and the application responsible for the data becomes flagged. An evident limitation of dynamic analysis is that it suffers from run time overhead, where the process may take more time than it needs to.

C. Machine Learning

Machine learning is a prominent pathway for researchers in detecting malware due to its ability to automatically classify Android applications. Unlike static analysis and dynamic analysis, which requires humans to manually input and make decisions for them, machine learning algorithms can take in datasets and teach themselves to make accurate predictions of later input data.

An example of machine learning algorithms that is widely used for malware detection, is deep learning. Inspired by the human brain, deep learning algorithms classify applications by connecting neurons to other neurons. These neurons pass a message or signal to other neurons based on the received input and form a complex network that learns with some feedback mechanism. Current classifiers that use deep learning like AdversarialDroid [6], DroidDetector [7], DL-Droid [15], employ several layers that connect to each other and produce a final result to the output layer. While deep learning is proven to perform better than traditional techniques, its disadvantages is that it requires a large amount of data for training, which is computationally expensive, in order to do better than other machine learning techniques. Deep learning based approaches are also prone to adversarial example attacks. In addition, there were many other research efforts that also used deep learning to help detect malware [19, 20].

A machine learning algorithm that will be used in this research paper is the support vector machine. This algorithm uses a hyperplane and support vectors to help classify Android applications as benign or malicious. A hyperplane is a decision boundary that is N-dimensional. The more features an application has, the more dimensions the hyperplane becomes. Li and others [9] extracts risky API calls, permissions, and URLs and sends these features that correspond to widely accepted measures like TF-IDF to a SVM model that produces a result based on the hyperplane. Support vectors allow the margins that shape the hyperplane to be as fitting as possible to all the data points and find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. The maximum margin distance provides some reinforcement so that future data points can be classified with more confidence. While SVM is also more effective than traditional malware detection techniques, like deep learning, it is also prone to adversarial example attacks.

D. Adversarial Example Attacks

Adversarial example attacks are malicious applications that evade the detectors of traditional approaches such as static and dynamic analysis. Malware authors use generative methods that tweak and mutate existing malware to create new attacks. E-MalGAN [12] is an invasive method that does not require information about its target and plays a noncooperative game between the generator model and the discriminator model. Malware recomposition variation (MRV) uses semantic analysis of existing malware to create and mutate feasible malware [13]. It produces evolution and confusion attacks which mimic and automates the evolution of malware that can hide their malicious feature during the detection phase and cause the classifier to misidentify it as benign. Once the application passes through the classifier, an intruder can remotely install additional code that can harm and steal private information, making the attack successful. Li et al. proposed DroidEnemy, which is a malware detection approach that can battle against adversarial example attacks [21].

III. Methodology

In this section, we first introduce the overall architecture of the proposed approach. A detailed description of each part of the scheme will then follow to get a better understanding of the overall process of detecting malware. Figure 1 displays the overall structure of the proposed work.

A. Overview

As shown in Figure 1, there are three main components of EnsembleDroid: app decompilation, feature extraction, and classification. During the app decompilation stage, Android applications are unpacked into a set of files such as smali

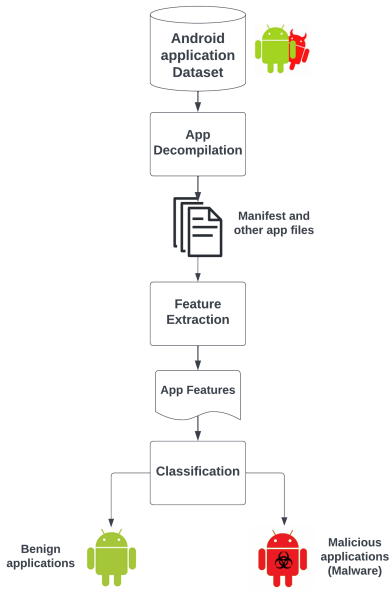

Fig. 1: Overview of Ensemble Learning Approach

files, source files, and the manifest file. The key features we will be using in this work will be the Android permissions that are extracted from the manifest file. The permissions will then be combined to create a feature set that will be used to train and test the stacked ensemble learning classifier model. Finally, the model produces a result of whether the Android application is malicious or benign.

B. Decompilation

All applications used on the Android operating system are in apk format. Within the apk file, there are features that make up the application such as the permissions it requests or the URLs it uses. In order to find these features, the apk file must be decompiled into different files which correspond to those features. In this work, we use an open source tool called Apktool [17], which could decompile the apk files into the readable manifest file and source files.

Apktool unpacks the apk file into a set of readable files, most notably the manifest (XML) file. The manifest file contains permissions that the application uses. These permissions contain the ”uses-permission” tag along with the label of what the permission is. For example, Figure 2 shows the permissions used by an application sample. The “android.permission.INTERNET” permission allows the application to access the internet and “android.permission.CAMERA” allows the application to access your device’s camera. In the interest of hackers, permissions like these can be extremely dangerous when not detected.

C. Feature Extraction

Once an apk file is unpacked and we are able to read the manifest file, we can then extract the permissions that the application uses. In order to do this, we develop a Python

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<permission android:name="a2z.Mobile.Event4264.permission.C2D_MESSAGE" android:protectionLevel="signature"/>
<uses-permission android:name="a2z.Mobile.Event4264.permission.C2D_MESSAGE"/>
<uses-permission android:name="android.permission.BLUETOOTH" android:required="false"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" android:required="false"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

Fig. 2: Depiction of permissions inside a manifest file

program that is used to open up the manifest file and create a feature vector based on the permissions that are requested in the manifest file. Each feature vector is represented as a single array of 1’s and 0’s. The feature vectors are later combined to make one 2D array feature set to write to the classifier as a CSV file.

$$\begin{cases} 0, & \text{if } x \text{ exists in application} \\ 1, & \text{otherwise} \end{cases}$$

Fig. 3: Cases for Feature Vector

The cases for creating a feature vector are depicted above in Figure 3 where x represents a permission in the application. To create a feature vector, a list of the permissions used in an application is compared to the list of all Android permissions. We iterate through the application’s permission list and check whether it exists in the list of all Android permissions. If a specific permission is requested by a given Android application, it will receive an 1, and 0 elsewise. Finally, the last column of an app’s feature vector will represent if the application is malicious or benign: it will be assigned the label of 1 if it is malicious or 0 if it is benign, in order to efficiently keep track of data for training and testing.

D. Classifier

After the feature set is written to a CSV file, the data will be evaluated by the classification model. The stacked ensemble learning classifier consists of two components: the base models and the meta-model. The base models consist of several weak learners. While these learners produce accuracies just slightly better than random guessing, it is computationally fast to train when compared to strong learners. Having multiple weak learners take in data allows multiple predictions to be made. These predictions are served as input to train the meta-model and produce the best final prediction. The overall functionality of stacking is shown in Figure 4.

Rather than using different types of classifiers as weak learners, the homogeneous stacking model sticks to one single type of learner. This classifier uses multiple support vector machine classifiers as the base models. On the other hand, heterogeneous stacking involves a variety of weak classifiers. This approach is more popular because it allows the base models to use different classification algorithms for prediction.

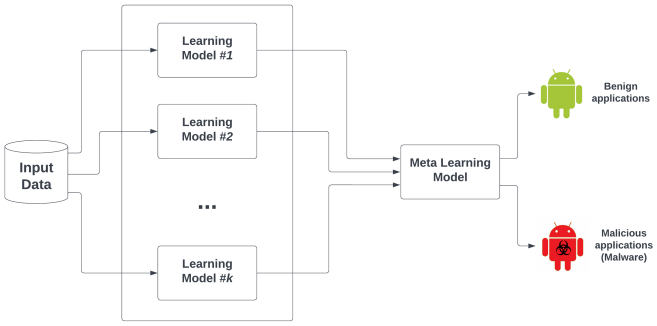

Fig. 4: Stacking Ensemble Learning Structure

IV. Experimental Study

In this section, we will discuss the experiments performed as well as the corresponding results that can be used to evaluate our proposed framework.

A. Dataset

In the experiment, we use Drebin [22] as the source for malware, which is one of the most widely accepted datasets of Android malware containing 5,560 applications from 179 malware families. The benign apps are privately collected from the Google Play Store which contains 1,550 applications. An 80/20 split was used to separate the data where 80% was used for training and the 20% was used for testing.

B. Experimental Results

For both homogeneous stacking and heterogeneous stacking models, data from the CSV file was split to ensure that the base models will be using the training set to produce their predictions and the metal-model will be using the testing set to assess its performance. A 10-fold cross validation approach is also used to train the meta-model.

```
# Defining homogeneous base model

level0 = list()
level0.append(('svm1',SVC(C=0.1, gamma = 0.1, kernel = 'rbf')))
level0.append(('svm2',SVC(C=0.1, gamma = 1, kernel = 'rbf')))
level0.append(('svm3',SVC(C=1, gamma = 0.1, kernel = 'rbf')))
level0.append(('svm4' ,SVC(C=1, gamma = 1, kernel = 'rbf')))
level0.append(('svm5', SVC(C=10, gamma = 0.1,kernel= 'rbf')))
level0.append(('svm6', SVC(C=10, gamma = 1,kernel= 'rbf')))
level0.append(('svm7',SVC(C=0.1, gamma = 0.1, kernel = 'linear')))
level0.append(('svm8',SVC(C=0.1, gamma = 1, kernel = 'linear')))
level0.append(('svm9',SVC(C=1, gamma = 0.1, kernel = 'linear')))
level0.append(('svm10', SVC(C=1, gamma = 1, kernel = 'linear')))
level0.append(('svm11',SVC(C=10, gamma = 0.1, kernel = 'linear')))
level0.append(('svm12',SVC(C=10, gamma = 1, kernel = 'linear')))

# Defining meta-model
level1 = MLPClassifier()

# Building stacking classifier
stack_clf = StackingClassifier(estimators = level0, final_estimator = level1, cv=10)
stack_clf.fit(X_train, y_train)
```

Fig. 5: Homogeneous Stacking Classifier

In homogeneous stacking (Figure 5), a total of 12 support vector machine classifiers were used as the base models of the homogeneous stacking classifier. Each SVM classifier had different hyperparameter values in order to make them unique.The most common hyperparameters chosen to tune were: C, gamma, and kernel. The values for each hyperparameter was defined based on the tuning of a default support vector machine classifier by using GridSearchCV, a cross validation method that produces the best values for given parameters. After the base models are set, we choose the multilayer perceptron classifier as the meta-model.

	precision	recall	f1-score	support
0	0.76	0.85	0.80	292
1	0.96	0.93	0.94	1052
accuracy			0.91	1344
macro avg	0.86	0.89	0.87	1344
weighted avg	0.91	0.91	0.91	1344

Fig. 6: Homogeneous Stacking Classification Report

The classification report in Figure 6 showed that the classifier was able to produce a 91% accuracy in predicting whether an application was malicious or benign. Along with the accuracy, a report on its precision, recall, and F1-score is presented. These additional metrics allow researchers to get a better understanding of the overall performance of the classifier.

```
# Building base models
level0 = list()
level0.append(('knn', KNeighborsClassifier(leaf_size=10,metric= 'euclidean', n_neighbors= 11, p= 1, weights= 'distance')))
level0.append(('svm', SVC(C=10, gamma = 0.1, kernel = 'rbf')))
level0.append(('lr', LogisticRegression(C=0.0001, max_iter= 100, penalty= 'none', solver='sag')))

# Building meta-model
level1= LogisticRegression()

# Building stacking model
stack_clf = StackingClassifier(estimators = level0, final_estimator = level1), cv=10)
stack_clf.fit(X_train, y_train)
```

Fig. 7: Heterogeneous Stacking Classifier

The heterogeneous stacking classifier depicted by Figure 7 consists of 3 different weak learners. Having the variety of learners allows the base models to work with several different algorithms and produce different predictions. The 3 chosen supervised machine learning algorithms used as the base models are: K-Nearest Neighbor, support vector machine, and logistic regression. Each base model was tuned using GridSearchCV to find their best values for their parameters. This is done to slightly improve each base model in order to give better predictions to the meta-model. The default logistic regression learner was chosen as the meta-model.

As seen by the classification report in Figure 8, the heterogeneous stacking classifier also had a 91% detection accuracy. Other measurements on the performance like precious, recall, and F1-score are displayed below.

V. Conclusion

In this paper, we proposed EnsembleDroid, a malware detection approach using ensemble learning. Specifically, we explored both homogeneous and heterogeneous stacking models when deploying the ensemble learning algorithm. When we build the homogeneous stacking classifier, we use multiple support vector machines as the base models and the

	precision	recall	f1-score	support
0	0.77	0.83	0.80	292
1	0.95	0.93	0.94	1052
accuracy			0.91	1344
macro avg	0.86	0.88	0.87	1344
weighted avg	0.91	0.91	0.91	1344

Fig. 8: Heterogeneous Classification Report

multilayer perceptron classifier. The heterogeneous stacking classifier uses the K-nearest neighbor, support vector machine, and logistic regression algorithms as the base models and logistic regression model again as the meta-model. During the experimental study, both models interestingly produced a 91

Now let us discuss the future research directions. First, due to the time limitation for this summer research program, the dataset is considered to be relatively small for the experimentation purpose. The ratio of benign to malicious applications could also be another factor for the further improvement to the accuracy of the classifiers. For future works, the dataset should be larger and contain a better ratio of benign to malicious apps. In addition, it would also be interesting to evaluate the performance of malware detectors against adversarial example attacks. As technology is getting advanced progressively, opportunities for malware to evade the detection of classifiers will surely be growing rapidly. Thus, this paper has set a good ground for further research in the security of smartphone devices.

VI. Acknowledgement

I would like to thank Dr. Wenjia Li, Dr. N. Sertac Artan, Dr. Ziqian Dong, and the community of the REU program at New York Institute of Technology for giving me the opportunity to conduct research in the area of malware detection in smartphone devices. I would also like to thank the National Science Foundation Grant No. CNS-1852316 for offering me this research opportunity.

References

[1] Published by Statista Research Department and J. 27, "Smartphone OS Market Share Forecast 2014-2023," Statista, 27-Jul-2022. [Online]. Available: https://www.statista.com/statistics/272307/market-share-forecast-for-smartphone-operating-systems/. [Accessed: 27-Jun-2022].

[2] Published by Statista Research Department and J. Johnson, "Global android malware volume 2020," Statista, 07-Jul-2022. [Online]. Available: https://www.statista.com/statistics/680705/global-android-malware-volume/. [Accessed: 30-Jul-2022].

[3] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," 2012 IEEE Symposium on Security and Privacy, 2012, pp. 95-109, doi: 10.1109/SP.2012.16.

[4] D. Wu, C. Mao, T. Wei, H. Lee and K. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on Information Security, 2012, pp. 62-69, doi: 10.1109/AsiaJCIS.2012.18.

[5] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," in Proceedings of 9th International ICST Conference on Security and Privacy in Communication Networks, 2013, vol.127, pp. 86-103, doi: 10.1007/978-3-319-04283-1_6.

[6] G. D'Ambrosio and W. Li, "AdversarialDroid: A Deep Learning based Malware Detection Approach for Android System Against Adversarial Example Attacks," 2021 IEEE MIT Undergraduate Research Technology Conference (URTC), 2021, pp. 1-5, doi: 10.1109/URTC54388.2021.9701615.

[7] Z. Yuan, Y. Lu and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," in Tsinghua Science and Technology, vol. 21, no. 1, pp. 114-123, Feb. 2016, doi: 10.1109/TST.2016.7399288.

[8] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification." In Proceedings of the 16th ACM conference on Computer and communications security, pp. 235-245, 2009, doi: 10.1145/1653662.1653691.

[9] W. Li, J. Ge and G. Dai, "Detecting Malware for Android Platform: An SVM-Based Approach," 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, 2015, pp. 464-469, doi: 10.1109/CSCloud.2015.50.

[10] W. Li, N. Bala, A. Ahmar, F. Tovar, A. Battu and P. Bambarkar, "A Robust Malware Detection Approach for Android System Against Adversarial Example Attacks," 2019 IEEE 5th International Conference on Collaboration and Internet Computing (CIC), 2019, pp. 360-365, doi: 10.1109/CIC48465.2019.00050.

[11] A. Salah, E. Shalabi, and W. Khedr, "A Lightweight Android Malware Classifier Using Novel Feature Selection Methods," Symmetry, vol. 12, no. 5, p. 858, May 2020, doi: 10.3390/sym12050858.

[12] H. Li, S. Zhou, W. Yuan, J. Li and H. Leung, "Adversarial-Example Attacks Toward Android Malware Detection System," in IEEE Systems Journal, vol. 14, no. 1, pp. 653-656, March 2020, doi: 10.1109/JSYST.2019.2906120.

[13] W. Yang, D. Kong, T. Xie and C. Gunter, "Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps," 2017 33rd Annual Computer Security Applications Conference, pp. 288-302, doi: 10.1145/3134600.3134642.

[14] G. D'Ambrosio and W. Li, "AdversarialDroid: A Deep Learning based Malware Detection Approach for Android System Against Adversarial Example Attacks," 2021 IEEE MIT Undergraduate Research Technology Conference (URTC), 2021, pp. 1-5, doi: 10.1109/URTC54388.2021.9701615.

[15] M. Alzaylaee, S. Yerima, S. Sezer, "DL-Droid: Deep learning based android malware detection using real devices," Computers Security, vol. 89, 2020, pp.1-11, doi: 10.1016/j.cose.2019.101663.

[16] "Apktool," Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. [Online]. Available: https://ibotpeaches.github.io/Apktool/. [Accessed: 12-Jun-2022].

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," ACM Transactions on Computer Systems, vol. 32, no. 2, pp. 1-29, doi: 10.1145/2619091.

[18] V. Moonsamy, J. Rong, and S. Liu, "Mining permission patterns for contrasting clean and malicious android applications," in Future Generation Computer Systems, 2014, vol. 36, pp. 122-132, doi.10.1016/j.future.2013.09.014.

[19] W. Li, Z. Wang, J. Cai and S. Cheng, "An Android Malware Detection Approach Using Weight-Adjusted Deep Learning," 2018 International Conference on Computing, Networking and Communications (ICNC), 2018, pp. 437-441, doi: 10.1109/ICCNC.2018.8390391.

[20] Z. Wang, J. Cai, S. Cheng and W. Li, "DroidDeepLearner: Identifying Android malware using deep learning," 2016 IEEE 37th Sarnoff Symposium, 2016, pp. 160-165, doi: 10.1109/SARNOF.2016.7846747.

[21] N. Bala, A. Ahmar, W. Li, F. Tovar, A. Battu, and P. Bambarkar, "DroidEnemy: battling adversarial example attacks for Android malware detection," in Digital Communications and Networks, 2021, pp. 1-8, doi: 10.1016/j.dcan.2021.11.001

[22] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. E. R. T. Siemens,"Drebin: Effective and explainable detection of android malware in your pocket," In Ndss, vol. 14, pp. 23-26. 2014.