AdversarialDroid: A Deep Learning based Malware Detection Approach for Android System Against Adversarial Example Attacks

Giuseppe D'Ambrosio, Wenjia Li
Department of Computer Science
New York Institute of Technology
New York, NY 10023
{gdambros, wli20}@nyit.edu

***Abstract*—As the predominant mobile operating system worldwide, Android suffers from various types of malware, which could cause severe security and privacy issues. To cope with them, many research efforts have been made to develop effective malware detectors. However, malware authors tend to evade detection by launching adversarial example attacks, in which Android applications could be mutated and thus causing confusion to malware detectors. In this paper, we propose a deep learning based approach to identify malware for the Android system in the presence of adversarial example attacks. To validate the proposed approach, we have conducted experimental study on real-world datasets.**

***Index Terms*—Android, malware, deep learning, adversarial example attack, evasion attack**

I. Introduction

In recent years, we have witnessed an explosive growth to the number of mobile users, and there are about 5.27 billion unique mobile users all around the world [1]. Of all of the mobile operating system, the Android system is the most popular one, which accounts for about 72 percent of the total market [2]. As the Android phones dominate the market alongside with its large collection of applications, the Android system has become very susceptible to malicious attacks, which are generally caused by malicious applications (malware).

As recently reported, there have been 482,579 new malware samples each month as of March 2020 [3]. The common goal for these malware samples is to acquire various sensitive user information and exploit it, which may include passwords, location information, banking information, and so on.

To battle the malware, three main categories of malware detection approaches have been studied, which are based on static analysis, dynamic analysis, and machine learning. Static analysis will detect or identify any issue with an application without executing it. Dynamic analysis could only detect malware when the applications are being executed. Either one of these analysis can be an essential component of the machine learning based approach. The machine learning based approach allows for smart detection of malware with the use of popular machine learning algorithms.

However, with the rise of these machine learning classifiers comes with the rise of adversarial attacks, which, in this case, are malicious attacks that appear to be benign and manage to get pass the classifier. There have been strides to conquer these adversarial attacks under machine learning, however, there have been proposals that a deep learning based approach could seem more effective and work much better.

There are many deep learning approaches such as Maldozer [9], DroidDetector [10], DroidDeepLearner [11], etc. Many of these methods generally utilize one of the deep learning algorithms, such as deep belief network (DBN), convoluted neural network (CNN), etc.

In this paper, we first explore the reason why the deep belief network would suffer from adversarial example attacks, and we then propose a way, by utilizing the deep belief network, to train our classifier to be more robust against these adversarial attacks. Finally, we use actual Android app datasets (which contain both malware and benign apps) to validate the proposed approach.

The rest of this paper will be organized as follows. Section II will present the related work that could help us get a better understanding of what leads up to our research. Section III will focus on the methodology which will display all the work needed for this procedure to take place, which is followed by the results and discussion in Section IV where we will discuss the science behind the network along side how we managed to get the classifier to detect these adversarial examples. Finally, in Section V, we will conclude our research and point out some possible future research directions that can be further explored on top of what have been accomplished in this paper.

II. Related Work

In this section, we will summarize some research efforts that were made in relation to tackling adversarial attacks in the machine learning format and how it leads us into the deep learning approach.

This research is partially supported by National Science Foundation Grant No. CNS-1852316.

A. Machine Learning based Approaches for Malware Detection

In recent years, various machine learning algorithms have been applied to detect malware for the Android system, such as support vector machine (SVM) [12], [13], clustering [14], [15], and random forest [16], [17].

Li et al. [4] conducts some research on how to make malware detectors more robust against adversarial attacks. In this work, they incorporate a mutated dataset to be processed so that the dataset then gets its features extracted and reduced. While this feature extraction process looks for API calls and requested permissions, it also extracts binary n-grams. Binary n-grams are essential to how this model can craft their adversarial examples. Data gets send to and from the classifier to train and be ready for adversarial attacks, depending on how well trained it is. The paper gave good insight of how these adversarial example attacks can be performed and the authors further speculated that deep learning algorithms could potentially be used to battle adversarial example attacks in the future.

B. Deep Learning based Approaches

As to what deep learning can offer from its machine learning counterpart, deep learning offers an extensible and exact solution for android malware characterization because it determines malware relying on patterns instead of signatures, a feature machine learning algorithms are known for. Not only that, deep learning makes great usage of neural networks for it to extrapolate on the essence of weighted biases from neurons. There are a variety of different deep learning algorithms as listed from [5], and the one that seems to be of common use is the deep belief network. The deep belief network is a generative graphical model, composed of multiple layers known as Restricted Boltzmann Machines, with the goal to classify data into any given category as indicated by the labeled data.

As stated from [5] along with the associated papers that it references, the deep belief network valued contribution is great when working with detecting malware, however, it can lead to performance suffering in the presence of adversarial example attacks.

C. Adversarial Attacks

Adversarial attacks are merely attacks into any deep learning model that an adversary has created for the sole purpose to cause the model to calculate errors. From [4], it states that adversarial attacks can branch off to two different sections, causative and exploratory. From the exploratory branch, the evasion attack is said to be one of the most popular kind of adversarial attack ever adopted. Likewise, implementation of the evasion attacks can take in many forms, and an official form will be describe in the later sections. However, to get more into the details of what an evasion attack does is merely in the name itself; evasion attacks aim to evade being detected from the classifier. An intruder exploits malicious occurrences during testing to have it be incorrectly classified as benign by a trained classifier, without having a collision over the training data. The intruder's scheme in this form builds up to rupture system solidarity, either with an objective or with a non selective attack according to it purpose.

III. Methodology

In order to fully layout the complexity of this research, this will be split into subsections tailoring the chronological order of how this can be implemented.

A. Dataset

When working with an investigation like this, it would be helpful to gather a dataset that can be resourceful to our needs. Android applications are packaged as .apk files, in which we need to eventually decompile in order to access its contents. There is an open source tool known as APKTools, which will be capable of doing just that. A particular file that we are looking for is the android manifest file, which is an .xml file that contains all the essential information that build our application. The file contain API calls and uses-permissions. For the sake of this research, we will be primarily focusing on acquiring the uses-permissions only of the .apk and there is research to back up why this is so in [6].

All the while understanding what data we want to look for, we now introduce the mendeley dataset [7]. This dataset contains more than 500,000 apk's, and its labeled features, known as the uses-permissions it uses during installation and even run-time. This dataset was also collected from three different sources, the google play store, a third party application downloader, and a given section dedicated solely to malicious apk's. This dataset is already extracted for us and is listed out in an .xlsx format which each row is a given .apk file and each column is a given feature, which can be denoted in a binary format; 0 meaning that this application is not using this feature, and 1 meaning that this application is using that given feature. For this research, there will be heavy attention on the google play store and the malicious data sets.

B. Feature Extractor

As such, when working with the ability to detect if an apk is malicious or not, it is to be expected that any apk that will pass though our system will be a novel apk, meaning that it's never been seen before by the system. However, before processing the apk though the system, it has to go under the extraction process in order for it to be read. As mentioned in the previous subsection, we begin to extract the apk's contents by a powerful tool known as apktools.

```
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<permission android:name="com.freesamplesus.myapp.permission.C2D_MESSAGE"
android:protectionLevel="signature"/>
<uses-permission android:name="com.freesamplesus.myapp.permission.C2D_MESSAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.RECORD_VIDEO"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
```

Fig. 1: An example of the Android manifest.xml file.

From there, we can examine the context of the android manifest file as shown in Figure 1, to which can be found towards the end of the file which contains all of the uses-permissions. In the next subsection, it will be revealed that python will be used in order to construct the deep belief network, so in algorithm 1 below, there is pseudo code that carefully formats our extracted uses-permissions into an excel sheet format. This will allow us to send forth our extracted data to our model for evaluation and possibly testing.

Algorithm 1: The algorithm that extracts the requested permissions from manifest file and writes them to the excel file.

```
wb = WorkBook();
sheet1 = wb.add_sheet('Sheet 1');
root = ET.parse("AndroidManifest.xml).getroot();
name = root.attrib["package"];
permission = root.findall("uses-permission");
i = 0;
for perm in permission do
    for att in perm.attrib do
        sheet1.write(i, 0, perm.attrib[att]);
    end
end
```

C. The Deep Belief Network

Most of the functionality of the deep belief network has been stated from Section II. For this deep belief network, it is adopted from a repository [8] so it can be possible for testing and training purposes. The project follows the same protocols and semantics by Geoffrey E. Hinton, the founder of the deep belief network. Implementation is easy as showcase in Figure 2 and in algorithm 2, where we can write up a reference to our excel file which contains all of the apk's from the mendeley dataset. We first introduce our dataset in a 2D array format. We will be working on dataset size consisting of 1000 benign and 1000 malicious apk's. There are approximately 400 features that are denoted, with the last column consisting of labeled data that specifies whether or not a given row, or in this case, an apk is benign or malicious. We will then partition our data and along that, instantiate our network, which is now our classifier, to indicate how many hidden layers, learning rate, number of epochs are needed. The data is then fitted by this classifier and then call in for predictions and then measure the performance.

```
# Splitting Data; 110,152 area we are working with
one_train, one_test, two_train, two_test = train_test_split(finalarray, a, test_size=0.2, random_state=0)

# Training
classifier = SupervisedDBNClassification(hidden_layers_structure=[256, 256],
                                         learning_rate_rbm=0.05,
                                         learning_rate=0.1,
                                         n_epochs_rbm=10,
                                         n_iter_backprop=100,
                                         batch_size=32,
                                         activation_function='relu',
                                         dropout_p=0.2)
#classifier.fit(X_train, Y_train)
classifier.fit(one_train, two_train)

two_pred = classifier.predict(one_test)

#Print out the Accuracy, F1-Score, Precision, and Recall
print('Done.\nAccuracy: %f' % accuracy_score(two_test, two_pred))
print('Done.\nF1 Score: %f' % f1_score(two_test, two_pred, average='weighted', zero_division='warn'))
print('Done.\nPrecision: %f' % precision_score(two_test, two_pred, average='weighted', zero_division='warn'))
print('Done.\nRecall: %f' % recall_score(two_test, two_pred, average='weighted', zero_division='warn'))
```

Fig. 2: Schematics of the Deep Belief Network format

Algorithm 2: The algorithm that collects and then sorts data into Dataframe.

```
Function DataProcess (a, b, c, d);
Input  : Two string a and b, and range c and d
Output: Dataframe
wb = load_workbook(a);
ws = wb[b];
data_rows = [];
for row in ws[c : d] do
    data_cols = [];
    for cell in row do
        data_cols.append(cell.value);
    end
    data_rows.append(data_cols);
end
```

There are four performance metrics that are useful to look at and those are precision, recall, f1-score, and accuracy. These performance metrics helps us establish an understanding to how well effective our classifier can be by the data that we give it for training and under testing. From this step, it is safe to say that our classifier can successfully detect benign and malicious applications.

D. Evasion Attacks

As cited from Section II, evasion attacks are a form of adversarial attacks that aims to bypass the classifier by any means necessary. There are a plethora of crafting such an attack yet one form of this attack that can be of useful investigation is the label flipping method. Label flipping is what the name is suggesting; simply flip the binary labels from our dataset to make the application appear to be benign when in reality it is malicious. There are two approaches to label flipping and they are:

- From a carefully selected row that represents a given malicious apk from our mendeley dataset, modify certain elements that can have the possibility to render as benign under testing or
- From the labeled data towards the end of the row that denotes whether or not a given apk is benign or not, selectively flip certain malicious denoted apk's to appear benign.

As such, the second approach will be more feasible to incorporate as it is comparatively easier than the former. From our dataset, what specifically differs from our malicious dataset and from our benign dataset is merely what features it uses, yet these features can appear in almost in a pattern-like format.

In Figure 3, the labeled data is shown at the last column which is in bold. To the right is Figure 4, which list out the same dataset but in this case, the label's are flipped while to the right of it is the original labeled data. Each and every row will be determined from our classifier and if guess incorrectly, that apk is part of the middle ground. This middle ground is

[[0,1,0,1,1,0,1,0,0,1,**1**], [[0,1,0,1,1,0,1,0,0,1,**0**], [1]

[0,1,0,1,1,0,1,0,0,1,**0**], [0,1,0,1,1,0,1,0,0,1,**0**], [0]

[1,1,0,0,1,1,1,1,0,0,**1**], [1,1,0,0,1,1,1,1,0,0,**0**], [1]

[0,0,0,0,1,1,0,0,0,1,**0**], [0,0,0,0,1,1,0,0,0,1,**1**], [0]

[1,1,1,0,1,1,1,0,0,0,**0**], [1,1,1,0,1,1,1,0,0,0,**1**], [0]

[0,0,1,0,1,1,1,0,0,1,**1**]] [0,0,1,0,1,1,1,0,0,1,**1**]] [1]

Fig. 3: Original Dataset Fig. 4: Mutated Dataset

what helps us note what labels have to be flipped in order for the classifier to perceive certain apk's to be benign. So in this case, we examine which malicious apk is located within this middle ground and flip its labeled data. Apk's that are eligible to be apart of the middle ground are the ones were noted as a false negative or a false positive and hold significance's as their patterns aren't known by the classifier.

From the image below, Figure 5, we can examine more closely into what this middle ground is. From our dataset, it is recorded the top five most frequently used features along side the app type; benign or malicious. It's these high frequencies like these that our classifier has difficulty distinguishing whats benign or malicious; an outlier some can say.

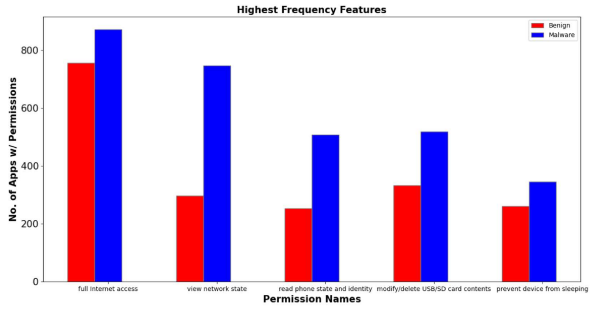

Fig. 5: Highest Feature Frequency amongst benign vs. malware

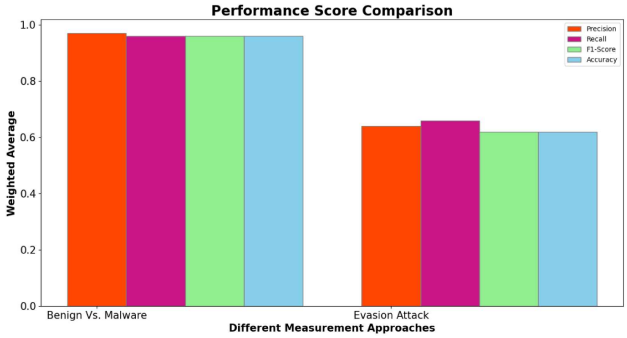

Fig. 6: The performance comparison of malware detector before and after launching evasion attacks.

From that understanding, we now take a look at Figure 6, which visualizes the performance metrics before and after the evasion attack occurs. It is clear that the attack effectively degraded the classifier and now we move forward to learning these types of attacks.

IV. Results and Discussion

This section will discuss and showcase the results of following works that were conducted during this research and give insight or details to which set up for a better understanding.

A. Detecting Adversaries

Moving forward from the previous section, there is an approach on how to circumvent evasion attacks like these. One method that is eminent is to retrain the classifier to detect these particular types of attacks. In retrospect, Figure 6 displays the performance of the malware detector with the added accuracy score of what it predicts is benign or malicious, yet the score isn't a perfect 100%. Reasons for this inaccuracies is due to the concept, as mentioned in the previous section, about the middle ground threshold that makes it difficult for the classifier to judge what is benign or malicious. How this task can be perform is to create adversarial example attacks to make the detector more robust, and that's by carefully selecting additional data from our mendeley dataset that falls under the same middle ground threshold and sent forth that additional dataset towards the classifier for the additional training. From there, it can be expected that any further evaluation that is coming from any adversarial attack will be prevented. Figure 7 showcases just that with an noticeable improvement in performance metrics after further training of these adversarial example attacks.

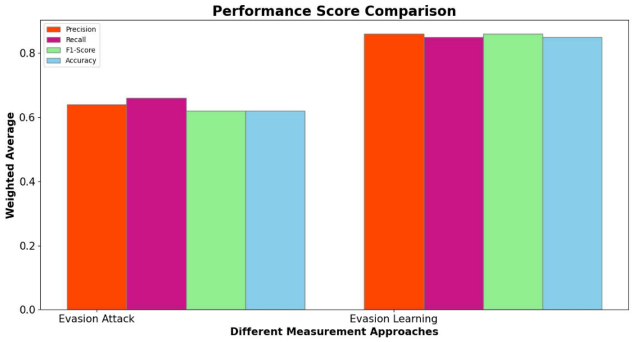

Fig. 7: The performance comparison of malware detector before and after additional training against evasion attacks.

Figure 8 lines up to how the performance is handled during different evasion attack sizes. It starts off at a reasonable pace, but towards understanding of the graph, it is shown that once it reaches 5000, it will remain difficult for it to learn more.

B. Deep Belief Network Performance Analysis

It is no doubt that the deep belief network can suffer from adversarial attacks. Though not mentioned from the previous section, a large amount of data, specifically an additional 5000 carefully selected apk's have gone into the additional

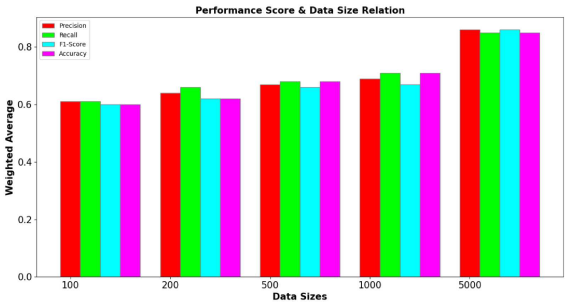

Fig. 8: The effect of dataset size on detection performance.

training for the deep belief network to detect such an attack. One possible idea for this lack of better performance could stem around the basic architecture the deep belief network is built on. The deep belief network dates back from 2006, and has been outperformed by other deep learning algorithms out there. One main component of interest could be the greedy pre-training step it takes when performed. This greedy step performs the reconstruction step of the inputted data that was sent in and follows up with a reconstruction error value to which gives a numerical value of how off it was when reconstructing the data. This reconstruction process is part of the greedy pre-training step and with that assumption, it can possible lead to the worst possible solution.

Now given that from the previous section there was discussion about the middle ground threshold, the deep belief network's greedy approach could possibly looked over the middle ground, as more data would needed to be required in order to examine the correct value fully. However, in the while, the deep belief network is the most used network when it comes to malware detection. What most of these other papers construct when incorporating the deep belief network is a merely hybrid network. What that can consist of is usually a mixture of other network's aspects or an edit to the network as it is, but never the less, the deep belief network on its own can't be reasonable feasible for the future of malware detection with the ever growing branch of adversarial attacks.

V. Conclusion and Future Works

In this paper, we examine the possibility of using the deep belief network to detect adversarial example attacks for Android system. From this paper, we established a coherent workflow for implementation and extraction of data towards the deep belief network classifier and list out its performance metrics of adversarial attacks or non-adversarial attacks. This paper has also gone over the logistics of the built up and/or architecture of the deep belief network that sets it to be flawed.

As for the future directions, we hope that this paper sets foot more onto the investigation of adversarial attacks and beyond. We also hope that there is development into the research of possible in-depth test that can help correlate to the understanding of these attacks and how they need to be approached when learning about them. Adversarial attacks are only going to get smarter and stronger as time continues and this paper aims to help bring out that purpose.

Acknowledgment

I would like to thank Dr. Wenjia Li, Dr. N. Sertac Artan, the REU program at New York Institute of Technology, and the National Science Foundation for making this research possible.

References

[1] Digital Around the World - DataReportal – Global Digital Insights. DataReportal. (n.d.). https://datareportal.com/global-digital-overview

[2] Mobile Operating System Market Share Worldwide. StatCounter Global Stats. (n.d.). https://gs.statcounter.com/os-market-share/mobile/worldwide.

[3] Johnson, J. (2021, January 25). Global Android malware volume 2020. Statista. https://www.statista.com/statistics/680705/global-android-malware-volume/.

[4] 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.

[5] Naway, Abdelmonim, and Yuancheng LI. "A Review On The Use Of Deep Learning In Android Malware Detection". Arxiv.Org, 2021, https://arxiv.org/abs/1812.10360.

[6] M. Kumaran and W. Li, "Lightweight malware detection based on machine learning algorithms and the android manifest file," 2016 IEEE MIT Undergraduate Research Technology Conference (URTC), 2016, pp. 1-3, doi: 10.1109/URTC.2016.8284090.

[7] Mahindru, Arvind (2020), "Android permissions dataset, Android Malware and benign Application Data set (consist of permissions and API calls)", Mendeley Data, V3, doi: 10.17632/b4mxg7ydb7.3

[8] Albertbup. (n.d.). albertbup/deep-belief-network. GitHub. https://github.com/albertbup/deep-belief-network.

[9] Karbab, ElMouatez Billah, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. "MalDozer: Automatic framework for android malware detection using deep learning." Digital Investigation 24 (2018): S48-S59.

[10] Yuan, Zhenlong, Yongqiang Lu, and Yibo Xue. "Droiddetector: android malware characterization and detection using deep learning." Tsinghua Science and Technology 21, no. 1 (2016): 114-123.

[11] Wang, Zi, Juecong Cai, Sihua Cheng, and Wenjia Li. "DroidDeepLearner: Identifying Android malware using deep learning." In 2016 IEEE 37th Sarnoff Symposium, pp. 160-165. IEEE, 2016.

[12] Arp, Daniel, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and C. E. R. T. Siemens. "Drebin: Effective and explainable detection of android malware in your pocket." In Ndss 2014, pp. 1-15.

[13] Li, Wenjia, Jigang Ge, and Guqian Dai. "Detecting malware for android platform: An svm-based approach." In 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, pp. 464-469. IEEE, 2015.

[14] Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: 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.

[15] Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. "Droidmat: Android malware detection through manifest and api calls tracing." In 2012 Seventh Asia Joint Conference on Information Security, pp. 62-69. IEEE, 2012.

[16] Alam, Mohammed S., and Son T. Vuong. "Random forest classification for detecting android malware." In 2013 IEEE GreenCom-iThings-CPSCom Conferences, pp. 663-669. IEEE, 2013.

[17] Zhu, Hui-Juan, Tong-Hai Jiang, Bo Ma, Zhu-Hong You, Wei-Lei Shi, and Li Cheng. "HEMD: a highly efficient random forest-based malware detection framework for Android." Neural Computing and Applications 30, no. 11 (2018): 3353-3361.