Preliminary Results of Applying Machine Learning Algorithms to Android Malware Detection

Matthew Leeds *, Travis Atkison †
Computer Science Department
University of Alabama
Tuscaloosa, AL
Email: * mwleeds@crimson.ua.edu, † atkison@cs.ua.edu

***Abstract*—As the use of mobile devices continues to increase, so does the need for sophisticated malware detection algorithms. The preliminary research presented in this paper focuses on examining permission requests made by Android apps as a means for detecting malware. By using a machine learning algorithm, we are able to differentiate between benign and malicious apps. The model presented achieved a classification accuracy between 75% and 80% for our dataset and the best combination of parameters. Future work will seek to improve the model by expanding the training dataset, taking more features into account, and exploring other machine learning algorithms.**

***Keywords*-Malware Detection, Android, Machine Learning.**

I. Introduction

Mobile devices are quickly becoming the primary means of communication and information access for billions of people. As such, they have become the target of many attackers who seek to gain access to these devices. In 2014, 16 million mobile devices were infected by malware. [1] These infections can lead to things such as identity theft, extortion, or robbery. For that reason, mobile device security is more important than ever. Due to limited computational resources and different execution environments, mobile security offers its own challenges not encountered with desktops. Recently, advancements in computer science and hardware improvements have made it possible for machine learning to effectively be applied to extract useful information from large datasets.

A. Android is the Target

The Android Operating System, which runs on 1.4 billion devices [2], [3], is the target for the vast majority of mobile malware, as noted in Figure 1. One of the main reasons is the fact that applications do not have to come directly from the Play Store. They can be "side-loaded" from alternative markets or directly installed from APK (Android Application Package) files. This means that no matter how good Google's vetting process is for the Play Store, malware can still be installed on a user's device. For iOS users, the applications must come directly from the official App Store; therefore, it is much more difficult for an attacker to get a malware app past the vetting process of the App Store.

Another reason that Android devices are targeted is the amount of Android devices in use. These devices are so numerous that even relatively low infection rates translate to a large number of actual infected devices. Furthermore, the vast majority of these devices are not up to date. According to [4], over 90% of Android devices run outdated versions of the operating system. For comparison, roughly 80% of iPhones are not on the latest version of iOS. [4] Updates are created to fix security holes that are discovered in the operating system that can be exploited by attackers. By not keeping the operating system up-to-date, the device is left vulnerable to potential attackers.

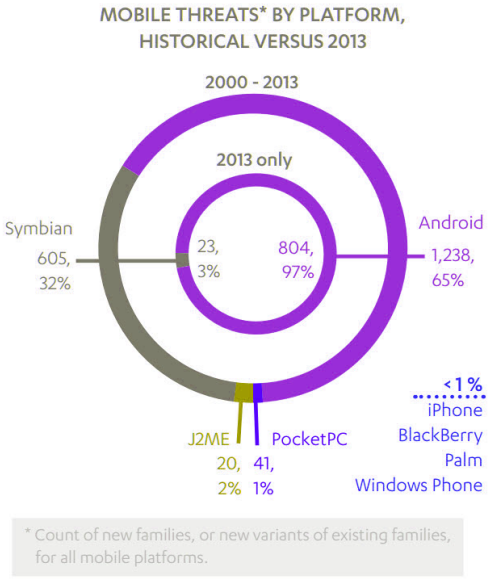

Fig. 1. This pie chart shows the proportion of mobile malware on each platform in 2013, and historically. [2]

B. Types of Android Malware

There are many different types of malware. Some times legitimate apps can have malware injected into them that will run in the background while the app is performing its intended function. The user may never know that the malware is actually running and performing unauthorized actions. Malware can also be downloaded at run-time. This is known as an "update attack". Other times the user can be enticed to download malware advertised for another purpose. This is known as a "drive-by download". [5]

Malware is written to exploit a variety of vulnerabilities and for a variety of purposes. Attackers can use malware to collect information such as contacts or passwords. They can also use malware to gain control of the device. By gaining this control, they can then send text messages to premium-rate numbers to steal money or add the phone to a botnet that can be controlled (for a DDoS attack for example). [5]

C. Existing Anti-Malware Strategies

There are several existing strategies to help avoid a malware infection. The first is to not root the device. Installing only reputable apps from the Play Store is also important. These apps are less likely to have malware injected into them because they are vetted before being allowed onto the Play Store. Another strategy is to not allow apps to be installed over USB. This is important because the USB drive itself could be infected and the apps installed on them are not necessarily vetted. Furthermore, it is important not to follow any sketchy links. As seen in many email virus, attackers can use links to install a virus on a computer. The same is true for mobile devices. In addition to these proactive measures, reactive measures, such as regularly scanning the device with anti-virus software, can help to further reduce the risk of installing malware on an Android device. While these strategies are good and help to avoid most malware, they are not a catch-all. More research needs to be done to develop new methods for detecting malware before it is even installed.

D. Machine Learning Strategies

As with everything in Computer Science, there is no "sliver bullet" machine learning algorithm that works for every problem. Generally, training time and accuracy are considered important metrics to consider. For the research presented in this paper, a simple neural network with a gradient descent optimizer function is used.

E. Literature Review

There have been several avenues of work in detecting malware on mobile devices using machine learning algorithms and techniques. These techniques range from Support Vector Machines (SVM) [6] to Neural Networks [7] to Classification Trees [8]. In [6], Sahs et al. described a malware detection method using a One-Class Support Vector Machine. Li et al. [9] used a SVM similar approach by looking at dangerous permissions that are likely used by Android malware. A neuro-fuzzy based clustering method is presented by Altaher et al. in [8]. Altaher et al. uses a fuzzy clustering method to determine the appropriate number of clusters again looking at permissions used by the Android application. They were able to refine their process in [10]. In [11], Alam et al. were able to use a random forest of decision trees in detecting malware in Android devices.

II. Methodology

To perform and automate the analysis on our Android data, a series of Bash and Python scripts were produced. These scripts handled everything from gathering Android app samples, processing the samples, training a model on the samples, testing the model, and graphing the resulting data. The scripts used for this project can be found at https://github.com/mwleeds/android-malware-analysis.

A. Gathering Malware Samples

Broadly, there are two strategies for gathering malware samples for research: get them "directly" using a honeypot or by purchasing them from black hat hackers, or acquire them from public or semi-public repositories intended for research. Andrototal.org is one such repository available to researchers [12], and it was used to gather the samples for the research presented in this paper. Once access to the provided API was obtained, the following command was used to download samples from the first three months of 2016:

```
$ python samples_cli.py getbydate
-at-key <API key> 20160101:0000
20160401:0000
```

Samples were also downloaded from 2013, 2014, and 2015. Because the evolution of malware can be significant over time, recency is a major concern for training an accurate prediction model (this is extremely true for any production use). The samples came as `.apk` files named with their own cryptographic hash to avoid confusion and confirmation of accurate download.

B. Classification

When developing a classification method, it is imperative to know whether or not each app in the training set is malicious in order to accurately train the model. The dataset used in this research came from Andrototal.org, which provides this needed information. Using the known classification category, the apps were sorted into folders, `malicious_apk` and `benign_apk`. We were able to use the following command to get the reports from AndroTotal based on the hash of the APK file:

```
$ python andrototal_cli.py analysis
-at-key <API key> + <hash of apk>
```

C. Feature Extraction

For this preliminary research effort, the permissions requested by apps was the main feature set that was taken into account. Since malicious apps often have different "permission signatures" than benign ones, it was felt that this is a good place to begin our investigation. In the future as this effort matures, more features will be taken into account.

There were several steps involved in extracting the features needed for this effort. The following are the main two aspects to retrieving the features. A Bash script was created and used to unpack the `apk` files into the appropriate directories. The script also converted the `AndroidManifest.xml` file from its original binary format to a plain text file; and lastly, the script checked to see if there was a valid XML. The names of all apps with valid XML files were put in a file.

Next, a Python script was created to read the names of the usable apps, examine each one's `AndroidManifest.xml`,

then traverse the tree to find `uses-permisssion` elements. For each of the standard permissions (app-specific permissions were ignored), the permissions presence for each of the apps was recorded as ether a 1 or 0. This effectively created a two dimensional array of bits. Lastly, this information, as well as each app's classification value, was stored in a JSON file.

D. Training the Model

```
# the first chunk of each will be used for testing (and the rest for training)
for i in range(int(sys.argv[1])):
    j = random.randrange(1, len(malicious_app_name_chunks))
    k = random.randrange(1, len(benign_app_name_chunks))
    app_names_chunk = malicious_app_name_chunks[j] + benign_app_name_chunks[k]
    batch_xs = [dataset['apps'][app]['vector'] for app in app_names_chunk]
    batch_ys = [dataset['apps'][app]['malicious'] for app in app_names_chunk]
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Fig. 2. This is the Python code that chooses chunks of apps using a random number generator and feeds them into the model so it can learn from them. The `train_step` variable tells TF to use gradient descent optimization to minimize the cross entropy (the difference between the correct and the actual results).

TensorFlow allows you to write a small amount of high-level code defining parameters and it takes care of the implementation. [13] The TensorFlow MNIST tutorial provided a useful guide for how to use TensorFlow for classification. Specifically, a neural network was created to link each input variable to each output classification. Gradient Descent optimization was used with a step size of 0.01 for the training. Batches of apps with equal proportions of malware and benign were run through the model many times so it could learn (update the weights). Figure 2 provides the Python code used to feed the model. Figure 3 provides a visual of the organization of the network. This provides the classification of the given inputs.

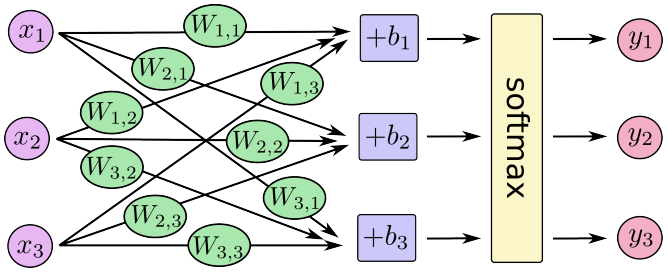

Fig. 3. This figure shows how a set of weights and biases followed by a softmax function can be used to classify a set of input values. Each weight can be thought of as a neuron and each value for y represents a category (in our case y1=1 means malicious and y2=1 means benign).

E. Evaluating the Model's Effectiveness

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

app_names_chunk = malicious_app_name_chunks[0] + benign_app_name_chunks[0]
test_xs = [dataset['apps'][app]['vector'] for app in app_names_chunk]
test_ys = [dataset['apps'][app]['malicious'] for app in app_names_chunk]
print(sess.run(accuracy, feed_dict={x: test_xs, y_: test_ys}))
```

Fig. 4. This is the Python code that evaluates the model's effectiveness by comparing its classifications of a set of apps with their actual classifications.

In order to show that the model created from the TensorFlow process described above is effective, the model must be tested on a set of samples that are **separate** from the samples that were used for the training. If the information stored in the weights generalizes beyond the training data, it's useful for classifying even never-before-seen malware such as 0-days. Equal-sized chunks of malicious and benign apps were input into the model, and its guess classification was compared with the correct answer, producing a number between zero and one that represents the fraction of samples classified correctly. Figure 4 provides the Python code that was used to evaluate the model's effectiveness. The Results section below will provide the values and an explanation of the results.

F. Running Trials

A Bash script was developed and used in order to run multiple experimental trials of the TensorFlow code for each number of training steps (1, 10, 100, 1000, 10000). The script also averaged the results, and then wrote the results to a CSV (comma separated value) file. This file could then be read by a Python script which used `matplotlib` to plot the resulting data.

III. Results

It is beneficial to run the same samples through the model multiple times to reinforce the resulting weights. Figure 5 below displays the resulting classification accuracy for the different number of training steps used. The training steps were increased up to 80 steps. As can be seen, the results plateau after n = 20. This means that for this data set the maximum learning threshold has been obtained; therefore, nothing more can be discovered from the data. Since the samples were split into 20 subsets, fewer than 20 training steps would not allow the model to learn from all the available data.

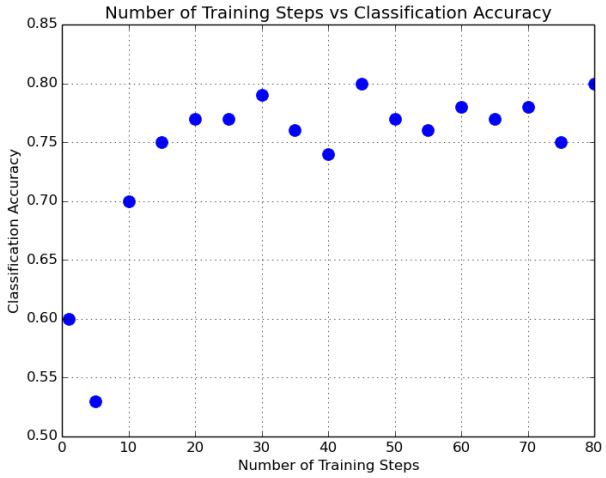

Fig. 5. This graph shows the model's classification accuracy after increasing numbers of training steps (where each "step" consists of running a subset of samples through the model). A score of 1.0 on the y axis would mean every sample in the test set was correctly classified as benign or malicious. Each data point is an average of 10 trials to reduce noise.

The dataset that was used in these experiments to produce these graphs contained 3,314 samples, of which 2,444 were benign and 870 were malicious. The relatively large dataset allowed the model to develop an accurate, generalized classification function rather than focused learning just around the training data. When using a smaller dataset with the same model, the highest classification accuracy that was achieve was around 70%.

As you can see in Figure 6, running samples through the model too many times severely reduces the classification accuracy. This is due to the phenomenon of over-fitting, meaning that the model becomes attuned to the features in the training data and is unable to generalize to other samples it has not seen before.

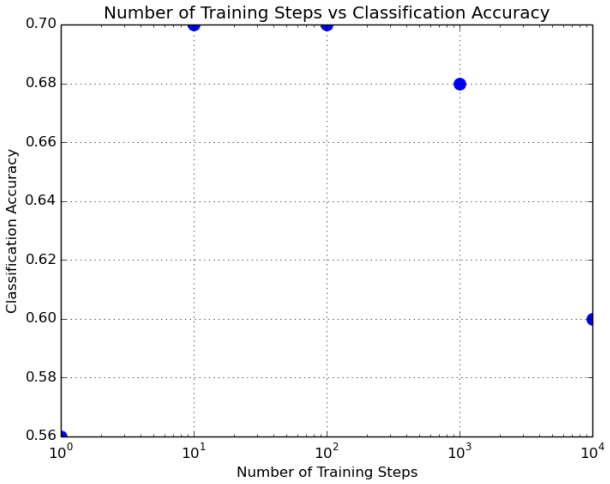

Fig. 6. This graph shows how the model's classification accuracy quickly drops off after exponentially increasing numbers of training steps.

IV. Future Improvements

While the work presented in this paper focuses only on permission requests made by android apps, future work would expand the number of features used to train the dataset. A dynamic analysis of the app's behavior in a VM could provide even more useful data (system calls, for example). Since computational resources are not a concern, a larger dataset and more training steps could also help improve the model. However, we would like to be able to create an app that could be loaded on an Android device to run real time checks of the device. As such, the algorithm would need to be reparameterized to work with the limited resources of a phone. Finally, more advanced machine learning strategies could be explored and developed to help improve the results.

V. Conclusion

As the use and proliferation of mobile devices continues to increase, the need for sophisticated malware detection methods is becoming more and more important. This paper presented a preliminary work focusing on developing an Android malware detection technique based on examining permission requests that were by Android applications. The model presented was able to achieve a classification accuracy rate of 80% on a dataset of over 3000 Android applications.

References

[1] L. Spencer. (2015) 16 million mobile devices hit by malware in 2014: Alcatel-lucent. [Online]. Available: http://www.zdnet.com/article/16-million-mobile-devices-hit-by-malware-in-2014-alcatel-lucent/

[2] G. Kelly. (2014) Report: 97% of mobile malware is on android. this is the easy way you stay safe. [Online]. Available: http://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/#4ddd7dbe7d53

[3] J. Callaham. (2015) Google says there are now 1.4 billion active android devices worldwide. [Online]. Available: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide

[4] D. Security. (2016) Duo security finds over 90 percent of android devices run outdated operating systems. [Online]. Available: https://duo.com/about/press/releases/duo-security-finds-over-90-percent-of-android-devices-run-outdated-operating-systems

[5] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[6] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Intelligence and Security Informatics Conference (EISIC), 2012 European*. IEEE, 2012, pp. 141–147.

[7] A. Altaher and O. BaRukab, "Android malware classification based on anfis with fuzzy c-means clustering using significant application permissions."

[8] A. Altaher, "Classification of android malware applications using feature selection and classification algorithms," *VAWKUM Transactions on Computer Sciences*, vol. 10, no. 1, pp. 1–5, 2016.

[9] W. Li, J. Ge, and G. Dai, "Detecting malware for android platform: An svm-based approach," in *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*. IEEE, 2015, pp. 464–469.

[10] S. Abdulla and A. Altaher, "Intelligent approach for android malware detection," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 9, no. 8, pp. 2964–2983, 2015.

[11] M. S. Alam and S. T. Vuong, "Random forest classification for detecting android malware," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 663–669.

[12] F. Maggi, A. Valdi, and S. Zanero, "Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors," in *Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM, November 2013.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/