A Comparison of Features for Android Malware Detection

Matthew Leeds
University of Alabama
P.O. Box 870290
Tuscaloosa, Alabama 35487
mwleeds@crimson.ua.edu

Miclain Keffeler
University of Alabama
P.O. Box 870290
Tuscaloosa, Alabama 35487
mkkeffeler@crimson.ua.edu

Travis Atkison*
University of Alabama
P.O. Box 870290
Tuscaloosa, Alabama 35487
atkison@cs.ua.edu

ABSTRACT

With the increase in mobile device use, there is a greater need for increasingly sophisticated malware detection algorithms. The research presented in this paper examines two types of features of Android applications, permission requests and system calls, as a way to detect malware. We are able to differentiate between benign and malicious apps by applying a machine learning algorithm. The model that is presented here achieved a classification accuracy of around 80% using permissions and 60% using system calls for a relatively small dataset. In the future, different machine learning algorithms will be examined to see if there is a more suitable algorithm. More features will also be taken into account and the training set will be expanded.

KEYWORDS

Malware Detection, Android, Machine Learning

ACM Reference format:
Matthew Leeds, Miclain Keffeler, and Travis Atkison. 2017. A Comparison of Features for Android Malware Detection. In *Proceedings of ACM SE '17, Kennesaw, GA, USA, April 13-15, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3077286.3077288

1 INTRODUCTION

Billions of people are using mobile devices as their primary means of communication and information access. Research estimates that there will be more than six billion smartphone users by 2020 and securing these devices should be a top priority both in business and personal use [26]. Nonetheless, these devices have become the target of many attackers who seek to gain access to them. It was estimated that malware had infected 16 million mobile devices as of 2014 [27]. Malware is extremely dangerous and can give the attacker access to the user's personal information such as passwords, bank accounts, and other identity information. Furthermore, it has become common practice for apps to record more data than is necessary, thus causing an oversharing problem as exemplified in [19]. As of this report, 82 percent of all apps know when Wi-fi and data networks are being used, when the device is turned on, and the current and last location of the device just from using the app [19]. In the first half of 2016, the GozNym malware took 4 million dollars in just days from 24 U.S. and Canadian banks by targeting customer accounts [26]. Malware is a part of our daily lives. Therefore, securing mobile devices and being able to detect when/if malware is being installed on a mobile device is of utmost importance. Due to limited computational resources and different execution environments, mobile security offers its own challenges not encountered with desktops. However, recent improvements in hardware and advancements in computer science have opened the door for machine learning algorithms to be effectively applied to extract useful information from large datasets. With the advancement of Tensorflow, a flexible open source software library for machine learning, it has become feasible to gain insights from data that wouldn't otherwise be possible. It is important to note that the research presented in this paper is an extension of a previous effort described in [14].

1.1 Targeting the Android Operating System

As seen in Figure 1, the Android Operating System is being targeted by numerous attackers. The mobile malware threats for Android devices far outnumber all the other platforms. Since 2010, SophosLabs has observed more than 1.5 million samples of Android malware [26]. One reason is the prevalence of the Android operating system. According to [8, 13], approximately 1.4 billion devices are running the Android operating system, and over 90% are using an out-dated version of the operating system [24]. Security holes in the operating system are used by attackers to gain access to the device. When these security holes are discovered, fixes are created and updates are released. If the operating system is not kept up-to-date, the device can be exploited by attackers. Recently, a number of vulnerabilities were found in the Android operating system. This collection of bugs was referred to as "Stagefright", in reference to the Stagefright libraries (underlying code in the OS that is shared by many applications) contained in the Android OS [20]. These vulnerabilities are particularly nasty due to the fact that they allow an attacker to remotely execute code on someone's phone by sending a specially crafted MMS message [20]. In fact, remote access tools (RAT), such as those used in Stagefright, are easily available on the Internet for sale. One particular tool even

*Corresponding author.

ACM SE '17, April 13-15, 2017, Kennesaw, GA, USA

DOI: http://dx.doi.org/10.1145/3077286.3077288

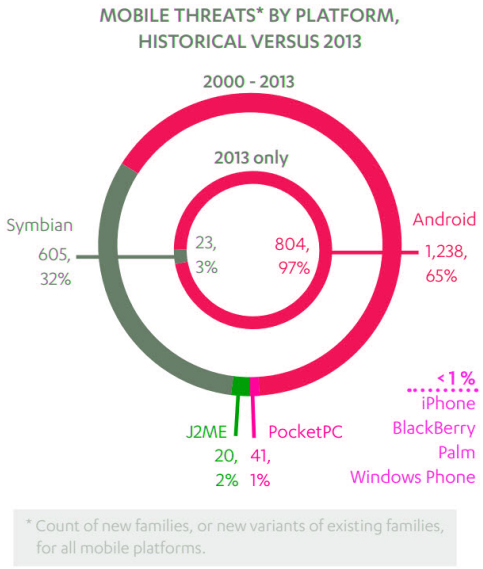

Figure 1: This pie chart shows the proportion of mobile malware broken down by individual platforms in 2013, and historically.[13]

has a well-polished website with tutorials, multiple pricing models and easy to use payment systems[20]. Because there are so many Android devices in use today, even relatively low infection rates translate to a large number of actual infected devices.

Another reason that Android devices are being targeted is the fact that users can download applications from third parties or load them directly onto the device from Android Application Files. They are not limited to just the Play Store. According to [20], of 150 million apps that were scanned on the Play Store in the last 3 months in over 190 countries, there were at least 37 million counts of malware detected. In the fourth quarter of 2015, there were 2.4 million new mobile malware threats detected [20]. However, following the events of the summer of 2015 (Stagefright), Google has committed to rolling out updates on a monthly basis for the foreseeable future [20]. iOS users are much less likely to have malware accidentally installed on their device because Apple required its users to get apps from the official App Store [26].

1.2 Android Malware

Malware can take on many different forms, and can be loaded on a device in many different ways. Some are trojan horse. They are attached to legitimate apps, but will perform malicious activities in the background without the user's knowledge or consent. As far as the user knows, the app is only performing its intended functions. Others are known as "drive-by" downloads [31]. These are downloads that the user is enticed to download that seem legitimate, but are actually just malware. Malware can also be downloaded at run-time. This is known as an "update attack".

Malware is written to exploit a variety of vulnerabilities and for a variety of purposes. Attackers can use malware to collect information regarding anything from contacts and passwords, to bank information and social security numbers. 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) [31].

1.3 Techniques to Avoid Malware

Avoiding malware is not always an easy task; however, there are several techniques or strategies that can be used to lessen the chances of a malware infection. One of the easiest is to limit app downloads to reputable apps from the Play Store. Since the Play Store has a vetting process that apps must go through before the Play Store will release them to the public, it is much harder for an attacker to get malware apps into the Play Store. Android devices also allow apps to be loaded from USB devices; however, this is not a good idea either. Inserting an infected USB drive into the Android device can itself infect the device, and any app installed on the USB drive may be infected. Since it did not come from the Play Store, it is unlikely that the app will have been tested for malware. Recently, Mcafee came out with a report which entails the most recent mobile threats, including those that were released in Stagefright and Stagefright 2. McAfee [20] recommends turning off your phone's ability to automatically retrieve MMS (Multimedia Message Service) messages. Furthermore, it is important to not follow links from unknown sources. As seen in many email viruses, 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.

1.4 Machine Learning Strategies

As with everything in Computer Science, there is "No Free Lunch" [30] meaning there is no machine learning algorithm that works for every problem. Generally, training time and accuracy are considered important metrics to consider. There are a lot of factors to consider when choosing the right machine learning algorithm for what you are doing. In [21] Williams et al. compare five such algorithms. One example is Naïve Bayes Tree (NBTree) which is a hybrid of a decision tree classifier and a Naïve Bayes classifier. It is designed to allow accuracy to scale up with increasingly large training datasets, which is something that would be beneficial to the research presented in this paper [21]. However, the authors conclude that while it did have one of the highest accuracies among the other algorithms tested, it had by far the highest normalized build times [21]. This could be a factor worth

considering depending on the application of said algorithm. For the research presented in this paper, a neural network with a gradient descent optimizer function is used.

1.5 Background

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) [23] to Neural Networks [5] to Classification Trees [4]. In [6], Zami et al. describes a framework that takes android apps and extracts permissions from each followed by classification of each as malware or goodware, somewhat similar to our process. However, the author then proceeds by clustering them through the K-Means clustering algorithm, followed by classification of each through the J48 Decision Tree algorithm. In a slightly different example, [12] Fereidooni et al. proposed ANASTASIA, a Machine Learning-based malware detection using static analysis of Android applications. Their tool extracted as many informative features as possible from Android applications and was tested on several classification algorithms to determine which one would perform the best. Another example, [15] Cen et al. shows how to use a probabilistic discriminative learning model with decompiled source code as well as permission features. In [23], Sahs et al. described a malware detection method using a One-Class Support Vector Machine. Li et al. [16] 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 [4]. 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 [2]. Some are able to simplify the identification of similar malware by HTTP Traffic. In [18], Aresu et al. they are able to group mobile botnet families by analyzing the HTTP traffic they generate. With the algorithm they used, a small number of signatures can be extracted from the clusters, allowing it to achieve a good tradeoff between the detection rate and the false positive rate. In a more simple example, [25] Alam et al. uses a system that exposes code clones and detects both bytecode and native code Android malware variants. In [29], Wang et al. propose a behavior chain based method that can detect Android malware including privacy leakage, SMS financial charges, malware installation, and privilege escalation by using matrix theory. In [11], Dong-Jie et al. use a static feature-based mechanism to extract representative configuration and trace API calls for identifying the Android malware. In [9], Santanu Kumar et al. used Conformal Prediction as an evaluation framework during runtime for their SVM-based classification approach. In [3], Alam et al. were able to use a random forest of decision trees in detecting malware in Android devices. According to [7], Bengio et al. found that gradient descent may be inadequate to train for tasks involving long-term dependencies, such as consistently dangerous permissions.

In [10], Dimjaševic et al. use "maline" to orchestrate running applications in virtual devices, sending random events to them, and recording the system calls they make. We make use of the same software in this paper with a different machine learning algorithm.

2 METHODOLOGY

To execute the methodology for this effort, a set of Python and Bash scripts were developed. These developed scripts automated our analysis of the Android data. They controlled most of the processes from gathering Android app samples, processing the samples, training a model on the samples, testing the model, and graphing the resulting data.

2.1 Gathering Malware Samples

Malware samples can be gathered in three basic ways. First, they can be "directly" gathered using a honeypot. Honeypots are mechanisms that can be used to gather information about attack methods. Second, they can be gathered from research repositories. These repositories can be either public or semi-public. Finally, they can be purchased directly from black-hat hackers. The malware samples used in this research were gathered from Andrototal.org. Andrototal.org is a semi-public repository that is available to researchers [17]. Malware is currently evolving and the changes over time can be significant. Therefore, it is extremely important that the data used to train the prediction models is current. For this reason, the malware samples gathered were from 2013, 2014, 2015, and the first three months of 2016. Once access to the provided API was obtained, a command similar to the following was used to gather the samples:

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

These samples came as `.apk` files named with their own cryptographic hash to avoid confusion and confirmation of accurate download.

2.2 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>
```

2.3 Feature Extraction

To compare static and dynamic analysis of apps, we look at two features: permissions requested at install-time (such as access to location information or cameras) and system calls made at run-time (such as filesystem or network operations). Since malicious apps tend to occasionally request different

permissions and make different system calls than benign ones, these features allow us to classify apps with respectable accuracy.

There were several steps involved in extracting the features needed for this effort. The following are the main two aspects to retrieving the permission 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 the XML is valid. 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 either 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.

In order to determine the system calls made at run-time, we used "maline", which can be found in [22]. Some modifications were made to make it work with a virtual device running API Level 25. Using that and the tools provided by Google, each application was installed on a virtual device and executed while being sent random input events. The system calls made during execution were logged using "strace". That file was parsed to determine the number of times each system call was used. This information was then reduced to a bit vector denoting whether each call was used at all so it could be used with the same ML model as the permissions data.

2.4 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})
```

Figure 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 [1]. 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. The same mathematical model is used to train for both features, permissions and system calls. This provides the classification of the given inputs.

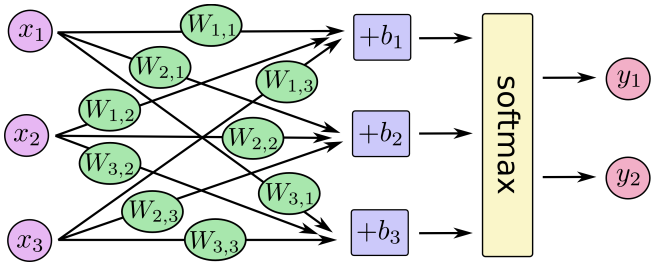

Figure 3: This figure shows how a set of weights and biases followed by a softmax function can be used to classify a set of input values. Each weight can be thought of as a neuron and each value for y represents a category (in our case y_1=1 means malicious and y_2=1 means benign) [28].

2.5 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}))
```

Figure 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 zero-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.

2.6 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 (up to 50 in intervals of 5). 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.

3 RESULTS

In order to verify the resulting weights, it is essential to run the experiment many times. It is also important to use the same sample set when testing the model with different numbers of training steps. Only by using the same sample set can a judgement on the best number of training steps be made. Figure 5 below displays the resulting classification accuracy for the different number of training steps used to examine the permissions data of Android devices for malware. The model was trained with 0-50 training steps in 5 step increments. Each "step" involved passing a sample subset through the model. If every sample in the test subset was correctly classified as either malicious or benign, then a score of 1.0 was recorded on the y-axis. Each data point on the graph was computed by running 10 trials and taking the average of the scores. This was done to verify the results and to help reduce noise. The results graph (Figure 5) shows that the model reached an average accuracy rate of approximately 80% with a high of 85% for 25 training steps. The accuracy rate plateaued quickly, reaching near peak accuracy after 5 training steps (n = 5).

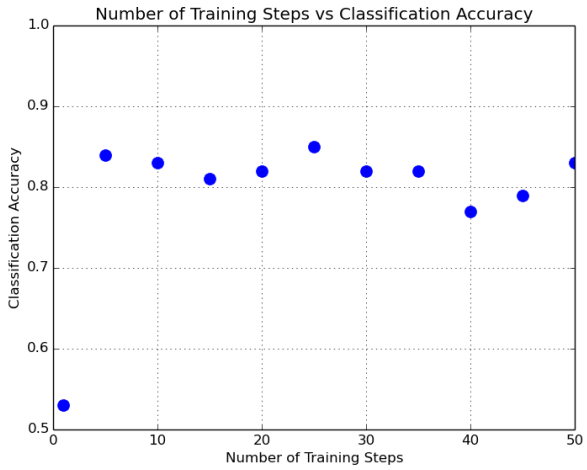

Figure 5: This graph shows the model's classification accuracy for different number of training steps when using the permissions data to find malware on Android devices.

Similar positive results were achieved for the system call features experiments, as shown in Figure 6. As with the permissions data feature model above, the system call data model was trained with 0-50 training steps in 5 step increments. Each "step" involved passing a sample subset through the model. Again, if every sample in the test subset was correctly classified as either malicious or benign, then a score of 1.0 was recorded on the y-axis. Each data point on the graph was computed by running 10 trials and taking the average of the scores. These results show that the models reached an average accuracy rate of approximately 60% with a high of 65% for 25 training steps. The accuracy rate plateaued again quickly, but took 10 training steps (n = 10) to reach near peak accuracy, which was slightly more than the permissions model.

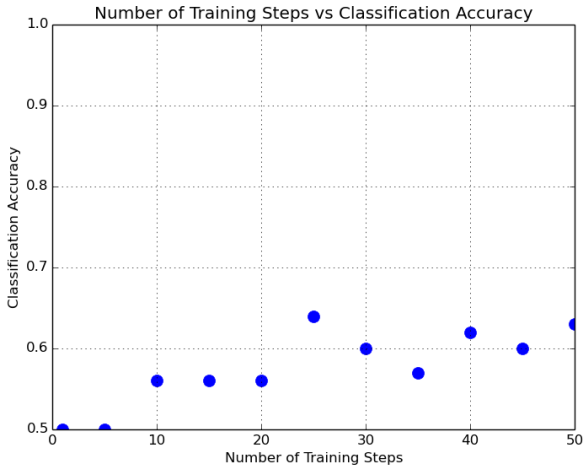

Figure 6: This graph shows the model's classification accuracy for different number of training steps when using the system call data to find malware on Android devices.

This fast plateau achievement on both methods is important to show that our models can detect malware quickly without having to waste time on additional training steps. This is promising for the future of this effort when moving toward more resource constrained environments.

From the figures above it is clear that the permissions were a better indicator of malicious activity than system calls. This is probably because permissions control access to sensitive data, whereas the same system call may be used innocently or maliciously depending on the parameters and the context. The low classification accuracy when using system calls could also be due to the number of samples used: 100 benign and 100 malicious.

4 FUTURE WORK

Android app permission requests and system calls are the main focus of this paper; however, future work could expand the number of features used to train the dataset and perhaps combine them into one model rather than training separately. Incorporating system call frequency (total number of times) rather than appearance (whether or not they occur) could further improve the model. Initially these ideas could be tested in a virtual machine environment where computational resources are not a concern; however, the ultimate goal is to create a downloadable Android app that could

run real-time checks for malware on the device. As such, the algorithm would need to be streamed-lined to work with the limited resources of a phone. Finally, other machine learning techniques can be tested to determine if there are more appropriate methods for malware detection.

5 CONCLUSION

Malware is a current threat facing Android users. As users have come to depend on these devices for communication and information, it is essential to make sure they are secure. Therefore, developing and testing new sophisticated malware detection techniques must be a priority. This paper compared two prominent features used to detect Android malware, permissions and system calls, and applied machine learning to both. The results showed that permissions data was better at detecting malware than system call data. An average classification accuracy rate of 80% was achieved when using permissions data to determine malicious activity on Android devices. Therefore, it is a reliable way to detect malware.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Shubair Abdulla and Altyeb Altaher. 2015. Intelligent Approach for Android Malware Detection. *KSII Transactions on Internet and Information Systems (TIIS)* 9, 8 (2015), 2964–2983.

[3] Mohammed S Alam and Son T Vuong. 2013. 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, 663–669.

[4] Altyeb Altaher. 2016. Classification of Android Malware Applications using Feature Selection and Classification Algorithms. *VAWKUM Transactions on Computer Sciences* 10, 1 (2016), 1–5.

[5] Altyeb Altaher and Omar BaRukab. 2015. Android malware classification based on ANFIS with fuzzy c-means clustering using significant application permissions. (2015).

[6] Zami Aung and Win Zaw. 2013. Permission Based Android Malware Detection. 2, 2 (2013).

[7] Simard Patrice Bengio, Yoshua and Paolo Frasconi. 1994. Learning Long-Term Dependencies with Gradient Descent is Difficult. (1994).

[8] John Callaham. 2015. Google says there are now 1.4 billion active Android devices worldwide. (2015).

[9] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. 2016. DroidScribe: Classifying Android Malware Based on Runtime Behavior. *Mobile Security Technologies (MoST 2016)* 7148 (2016), 1–12.

[10] Marko Dimjaševic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. 2015. Android malware detection based on system calls. *University of Utah, Tech. Rep* (2015).

[11] Te-En Wei Hahn-Ming Lee Kuo-Ping Wu Dong-Jie Wu, Ching-Hao Mao. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. (2012).

[12] Hossein Fereidooni, Mauro Conti, Danfeng Yao, and Alessandro Sperduti. 2016. ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications. In *New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference on*. IEEE, 1–5.

[13] Gordon Kelly. 2014. Report: 97% Of Mobile Malware Is On Android. This Is The Easy Way You Stay Safe. (2014).

[14] M. Leeds and T. Atkison. Preliminary Results of Applying Machine Learning Algorithms to Android Malware Detection. In *The 2016 International Symposium on MobileComputing, Wireless Networks, and Security (CSCI-ISMC)*.

[15] Luo Si Lei Cen, Christoher S. Gates and Ninghui Li. 2015. A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code. 12, 4 (July/August 2015).

[16] Wenjia Li, Jigang Ge, and Guqian Dai. 2015. Detecting Malware for Android Platform: An SVM-based Approach. In *Cyber Security and Cloud Computing (CSCloud), 2015 IEEE 2nd International Conference on*. IEEE, 464–469.

[17] Federico Maggi, Andrea Valdi, and Stefano Zanero. 2013. AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 49–54.

[18] Mansour Ahmadi Davide Maiorca Marco Aresu, Davide Ariu and Giorgio Giacinto. 2015. Clustering android malware families by http traffic. (October 2015).

[19] Mcafee. 2014. Mcafee Mobile Security Report: Whos Watching You? (February 2014).

[20] Mcafee. 2016. Mobile Threat Report: Whats on the Horizon for 2016. (2016). http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf

[21] Sebastian Zander Nigel Williams and Grenville Armitage. 2006. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. 36, 5 (October 2006).

[22] University of Utah Software Analysis Research Lab. 2015. maline. (2015). https://github.com/soarlab/maline

[23] Justin Sahs and Latifur Khan. 2012. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European*. IEEE, 141–147.

[24] Duo Security. 2016. Duo Security Finds Over 90 Percent of Android Devices Run Outdated Operating Systems. (2016).

[25] Ibrahim Sogukpinar Shahid Alam, Ryan Riley and Necmeddin Carkaci. 2016. DroidClone: Detecting Android Malware Variants by Exposing Code Clones. (2016).

[26] Sophos. 2016. When Malware Goes Mobile. (2016). https://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile.aspx

[27] Leon Spencer. 2015. 16 million mobile devices hit by malware in 2014: Alcatel-Lucent. (2015). http://www.zdnet.com/article/16-million-mobile-devices-hit-by-malware-in-2014-alcatel-lucent/

[28] TensorFlow. 2017. MNIST for ML Beginners. (2017). https://www.tensorflow.org/get_started/mnist/beginners

[29] Li Chenglong Yuan Zhenlong Guan Yi Wang, Zhaoguo and Yibo Xue. 2016. DroidChain: A novel Android malware detection method based on behavior chains. *Mobile Security, Privacy and Forensics, Pervasive and Mobile Computing* 32 (2016), 3–14.

[30] David H Wolpert and William G Macready. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation* 1, 1 (1997), 67–82.

[31] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 95–109.