A Comparison of Features for Android Malware Detection

Matthew Leeds *, Miclain Keffeler †, Travis Atkison ‡
Computer Science Department
University of Alabama
Tuscaloosa, AL
Email: * mwleeds@crimson.ua.edu,† mkkeffeler@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 examines two types of features of Android applications, permission requests and system calls, 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 of around 80% using permissions and 60% using system calls for a relatively small dataset. Future work will seek to improve the model by expanding the training dataset, taking more features into account, and exploring other machine learning algorithms.**

***Index Terms*—Malware Detection, Android, Machine Learning.**

I. Introduction

Mobile devices are quickly becoming the primary means of communication and information access for billions of people. Research estimates more than six billion smartphone users by 2020 and securing these devices should be a top priority both in business and personal use. [1] Nonetheless, 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. [2] 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. Furthermore, it has become common practice for apps to record more data than is neccesary, thus causing a oversharing problem as exemplified in [3]. As of this report, 82 percent of all apps know when you use Wi-fi and data networks, when you turn on your device, and your current and last location just from using the app. [3] In the first half of 2016, we saw the GozNym malware take 4 million dollars in just days from 24 U.S. and Canadian banks by targeting customer accounts. [1] Malware is a part of our daily lives. 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. With the advancement of Tensorflow, a flexible open source software library for machine learning, it has become very feasible to gain insights from data that wouldn't otherwise be possible.

A. Android is the Target

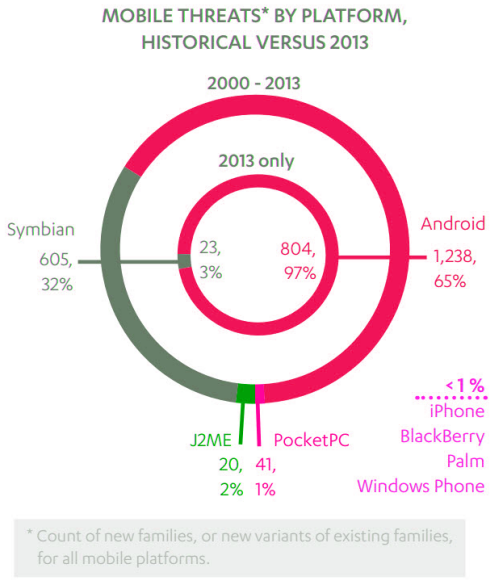

Fig. 1. This pie chart shows the proportion of mobile malware broken down by individual platforms in 2013, and historically. [4]

The Android Operating System, which runs on 1.4 billion devices [4], [5], is the target for the vast majority of mobile malware, as noted in Figure 1. Since 2010, SophosLabs has observed more than 1.5 million samples of Android malware. [1] One of the main reasons is the fact that applications do not have to come directly from the Play Store. According to [6], 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 over 6 months. In Q4 of 2015, there were 2.4 million new mobile malware threats detected. [6] These applications 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. However, following the events of the summer of 2015 (Stagefright), Google has committed to rolling out updates on a monthly basis for the forseeable future. [6] For iOS users, the applications must come directly

from the official App Store; therefore, it is much more difficult for users to inadvertently install malware. [1]

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 [7], 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. [7] 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. We saw some of the effects of this very recently, a number of vulnerabilities were found in the Android operating systems. 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. These vulnerabilities are particularly nasty due to the fact that they allow an attacker to remotely execute code on someones phone by sending a specially crafted MMS message. [6] In fact, remote access tools (RAT), such as those used in Stagefright, are easily available on the Internet for sale. One particular tool even has a well-polished website that puts most commercial software to shame with tutorials, multiple pricing models and easy to use payment systems. [6]

B. Types of Android Malware

There are many different types of malware. Sometimes 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". [8]

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). [8]

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. More recently, Mcafee came out with a report which entails the most recent mobile threats, including those taht were released in Stagefright and Stagefright 2. [6] reccommends turning off your phone's ability to automatically retrieve MMS (Multimedia Message Service) messages. Furthermore, it is important to not 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 "silver bullet" 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 [9] Williams et al. they compare five such algorithms. One example is Nave Bayes Tree (NBTree) which is a hybrid of a decision tree classifier and a Nave Bayes classifier. It is designed to allow accuracy to scale up with increasingly large training datasets (Something that would be quite beneficial to us). [9] However, they conclude that while it did have one of the highest accuracy among the other algorithms tested, it had by far the highest normalised build times. [9] This could be a factor worth considering depending on the application of said algorithm. 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) [10] to Neural Networks [11] to Classification Trees [12]. In [13], 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, he 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, [14] 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 the most performant one. 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 [10], 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 [12]. 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 [17].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 to achieve a good tradeoff between the detection rate and the false positive rate. In a more simple example, [19] Alam et al. uses a system that exposes code clones and detects both bytecode and native code Android malware variants. in [20], 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 [21], Dong-Jie et al. use a static feature-based mechanism to extract representative configuration and trace API calls for identifying the Android malware. In [22], Santanu Kumar et al. used Conformal Prediction as an evaluation framework during runtime for their SVM-based classification approach. In [23], Alam et al. were able to use a random forest of decision trees in detecting malware in Android devices. According to [24], Bengio et al. they found that gradient descent may be inadequate to train for tasks involving long-term dependencies, such as consistently dangerous permissions.

In [25], 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.

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 [26], 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

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 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.

In order to determine the system calls made at run-time, we used ”maline”, which can be found here [27]. 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.

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. [28] 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.

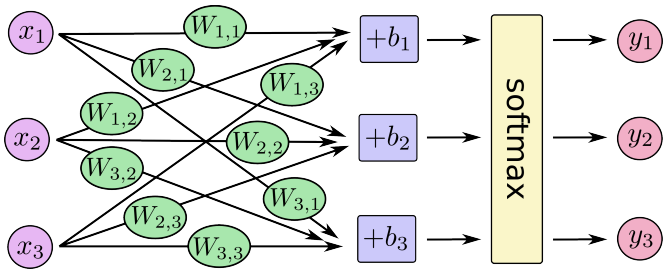

Fig. 3. This figure shows how a set of weights and biases followed by a softmax function can be used to classify a set of input values. Each weight can be thought of as a neuron and each value for y represents a category (in our case y1=1 means malicious and y2=1 means benign).

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 (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.

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 50 steps. As can be seen, the results plateau after n = 10. This means that for this data set and set of parameters the model can't learn anything more. Since the samples were split into 10 subsets, fewer than 10 training steps would not allow the model to learn from all the available data.

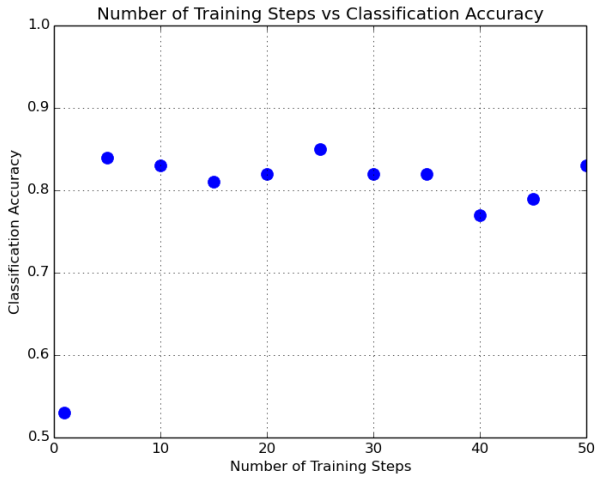

Fig. 5. This graph shows the model's classification accuracy after increasing numbers of training steps (where each "step" consists of running a subset of samples through the model) using the permissions data. A score of 1.0 on the y axis would mean every sample in the test set was correctly classified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

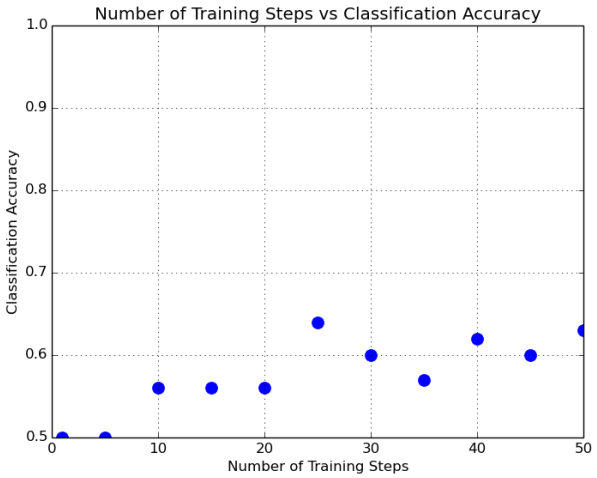

Fig. 6. This graph shows the model's classification accuracy using the system call data instead.

From the figures above it is clear that the permissions were a better indicator of maliciousness 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.

IV. Future Improvements

While the work presented in this paper focuses only on permission requests and system calls made by android apps, 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. 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 comparison of two prominent features used to detect Android malware, permissions and system calls, and applied machine learning to both. The model presented was able to achieve a classification accuracy rate of 80% using permissions data, indicating that they are a reliable way to detect malware.

References

[1] Sophos. (2016) When malware goes mobile. [Online]. Available: https://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile.aspx

[2] 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/

[3] Mcafee. (2014, February) Mcafee mobile security report: Whos watching you?

[4] 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

[5] 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

[6] Mcafee. (2016) Mobile threat report: Whats on the horizon for 2016. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf

[7] 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

[8] 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.

[9] S. Z. Nigel Williams and G. Armitage, "A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification," vol. 36, no. 5, October 2006.

[10] 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.

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

[12] 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.

[13] Z. Aung and W. Zaw, "Permission based android malware detection," vol. 2, no. 2, 2013.

[14] D. Y. Hossein Fereidooni, Mauro Conti and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications."

[15] L. S. Lei Cen, Christoher S. Gates and N. Li, "A probabilistic discriminative model for android malware detection with decompiled source code," vol. 12, no. 4, July/August 2015.

[16] 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.

[17] 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.

[18] M. A. D. M. Marco Aresu, Davide Ariu and G. Giacinto, "Clustering android malware families by http traffic," October 2015.

[19] I. S. Shahid Alam, Ryan Riley and N. Carkaci, "Droidclone: Detecting android malware variants by exposing code clones," 2016.

[20] L. C. Y. Z. G. Y. Wang, Zhaoguo and Y. Xue, "Droidchain: A novel android malware detection method based on behavior chains," *Mobile Security, Privacy and Forensics, Pervasive and Mobile Computing*, vol. 32, pp. 3–14, 2016.

[21] T.-E. W. H.-M. L. K.-P. W. Dong-Jie Wu, Ching-Hao Mao, "Droidmat: Android malware detection through manifest and api calls tracing," 2012.

[22] S. K. K. T.-M. A. J. K. Santanu Kumar Dash, Guillermo Suarez-Tangil and L. Cavallaro, "Droidscribe: Classifying android malware based on runtime behavior."

[23] 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.

[24] S. P. Bengio, Yoshua and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," 1994.

[25] M. Dimjaševic, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Android malware detection based on system calls," *University of Utah, Tech. Rep*, 2015.

[26] 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.

[27] U. of Utah Software Analysis Research Lab. maline. [Online]. Available: https://github.com/soarlab/maline

[28] 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/