Examining 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*—With the constantly increasing use of mobile devices, the need for effective malware detection algorithms is constantly growing. The research presented in this paper expands upon previous work that applied machine learning techniques to the area of Android malware detection by examining Java API call data as a method for malware detection. In addition to examining a new feature, a significant amount of work has been done in understanding how the model works and various ways of improving its accuracy. Ultimately a classification accuracy of around 80-85% was achieved using the JAVA API call feature.**

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

Submission Type: Regular Research Paper
Track: Computer Security
Corresponding Author: Travis Atkison (atkison@cs.ua.edu)

I. Introduction

Mobile devices are, for many people, the primary mode of communication and information access. Research believes that there will be over 6 billion smartphone users by 2020. As such, the security of these devices should be a top priority both in business and personal use [1]. Regardless, they have become the target of a plethora of attackers who want to obtain access to these mobile devices. In 2014, over sixteen million devices were held hostage by malware [2]. The effects of these infections can result in terrible outcomes such as extortion, identity theft, and even robbery. For this reason, mobile device security is more important than it has ever been. The limited computational resources and the multitude of different execution environments present a number of challenges in mobile security. Furthermore, it has become typical for applications to retain more data than is needed. This is causing an over sharing dilemma as shown in [3]. As of this report, 82% of all applications know if, and when, you use data networks and Wi-Fi, when you turn your device on, as well as your last and current location, all just from using the application [3]. In the first six months of 2016, the malware known as GozNym took $4 million in a few days from 24 banks, Canadian and U.S., by singling out customer accounts [1]. Malware is a piece of our day-to-day lives. Lately, advancements in Computer Science as well as hardware improvements have made it feasible for the use of machine learning to be applied effectively in the extraction of helpful information from larger datasets. Thanks to libraries such as TensorFlow, a flexible, high-level, open source framework for machine learning, it has become quite viable to gain insight from data that wouldn't otherwise be viable.

A. Android is the Target

Android devices are very popular. Their operating system runs on over 1.4 billion devices [4], [5]. It is the target for the sweeping majority of mobile malware. Beginning in 2010, SophosLabs observed over 1.5 million samples of malware, in Androids alone [1]. In fact, of 150 million apps that were tested on the Play Store over the course of 3 months (in over 190 countries), there were a minimum of 37 million occurences of malware detected [6]. For the duration of Q4 in 2015, over 2.4 million new mobile malware threats were detected [6]. Exacerbating the problem, these apps can be "side-loaded" from unofficial stores and markets or even directly installed from Android Application Package (APK) files. This essentially makes it so that no matter how good Google's malware detection and vetting process is for their store, malware can still be installed on a user's device. On the other side, for iOS users, their applications must come directly from the official App Store; as such, it is quite difficult for users to accidentally install malicious applications [1].

Android devices are so numerous that if a malware has a relatively low infection rate, this will still equate to a large amount of actual infected devices. In addition, the vast majority of Android devices are not up to date. According to [7], about 90% of these devices run older versions of their operating system. For comparison, nearly 80% of iPhones are not on the newest version of iOS [7]. Updates to these OS's are created, in many cases, to solve security problems that have been discovered in the operating system. More often than not, these can be utilized by attackers. By not keeping the Operating System updated, the device is left open to attackers. Quite recently, the effects of this were seen when a multitude of vulnerabilities were exploited in Android. The collection of vulnerabilities are referred to as "Stagefright" since that's the name of the core Android component that was exploited. These exploits are particularly nasty due to the fact that they permit an attacker to remotely execute code on a phone by sending a specially crafted MMS message [6]. After the initial exploit works, attackers will often try to retain control long-term via Remote Access Tools (RAT), which are easily available for purchase on the Internet. One particular tool has a well made

website with a multitude of pricing models, tutorials, and simple payment systems [6].

II. 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) [8] to Neural Networks [9] to Classification Trees [10]. In [11], 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, Fereidooni et al. [12] 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.

A probabilistic discriminative learning model is used by Cen et al. [13] with decompiled source code as well as permission features. In [8], Sahs et al. described a malware detection method using a One-Class Support Vector Machine. Li et al. [14] 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 [10]. 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 [15]. Some are able to simplify the identification of similar malware by HTTP Traffic.

Mobile botnet families are clustered by Aresu et al. by analyzing the generated HTTP traffic [16]. 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, Alam et al. [17] uses a system that exposes code clones and detects both bytecode and native code Android malware variants. In [18], 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 [19], Dong-Jie et al. use a static feature-based mechanism to extract representative configuration and trace API calls for identifying the Android malware. In [20], Santanu Kumar et al. used Conformal Prediction as an evaluation framework during runtime for their SVM-based classification approach. In [21], Alam et al. were able to use a random forest of decision trees in detecting malware in Android devices. According to [22], Bengio et al. found that gradient descent may be inadequate to train for tasks involving long-term dependencies, such as consistently dangerous permissions.

In [23], 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.

III. Feature Analysis

The work presented here is an extension of previous efforts [24]. In [24], feature selection was based on permissions and system calls. These features were input into a machine learning model. Specifically, it was a single layer neural network using a Gradient Descent Optimizer and softmax regression, implemented in TensorFlow. This method produced classification accuracy results of between 80-85% using permissions, and a considerably lower accuracy using system calls.

A. Examining Feature Weights

The `tensorflow_learn.py` script was modified to print the weights on each feature after the model finished training. Then another script was written to match the weights to human readable names and print them in descending order.

In the tables below, the learning rate was 0.01, the number of training steps was 50, and the number of apps in the dataset was 200. Higher weights indicate more certainty of maliciousness/benignity based on that feature, but since the weights for each feature go into a summing function before the activation function that decides how to classify the samples, they don't have to neatly fit into an interval of the Real numbers.

Table I
Permissions Indicative of Maliciousness

Rank	Permission	Weight
1	READ_PHONE_STATE	1.179
2	WRITE_APN_SETTINGS	0.938
3	INSTALL_PACKAGES	0.748
4	READ_SMS	0.611
5	GET_TASKS	0.579
6	RECEIVE_BOOT_COMPLETED	0.558
7	INTERNET	0.522
8	WRITE_SMS	0.497
9	MOUNT_UNMOUNT_FILESYSTEMS	0.429
10	BIND_ACCESSIBILITY_SERVICE	0.417

According to Table I, READ_PHONE_STATE is the #1 weighted permission by the network that indicates maliciousness. Upon further analysis, READ_PHONE_STATE ends up being the second most used permission in malicious applications. 912 of the 1,000 malicious applications used this permission, and the ratio of malicious to benign apps that used it was much higher than for the top permission (which was INTERNET). This helps to explain the heavy weight associated with this permission. This permission gives applications the ability to access information such as the phone number of the device and current cellular network information

[25]. It makes sense that malicious apps want the phone number, possibly to give to spammers. The SMS permissions (rows 4 and 8 of Table I) also make sense because a common way for black hat hackers to make money is by messaging paid numbers. The install packages permission, in row 3, can allow a device to be further infected by installing more malware.

Table II
PERMISSIONS INDICATIVE OF BENIGNITY

Rank	Permission	Weight
1	WAKE_LOCK	0.869
2	USE_CREDENTIALS	0.662
3	BLUETOOTH_ADMIN	0.571
4	ACCESS_NETWORK_STATE	0.558
5	BLUETOOTH	0.538
6	CALL_PHONE	0.479
7	PACKAGE_USAGE_STATS	0.470
8	DOWNLOAD_WITHOUT_NOTIFICATION	0.421
9	INTERACT_ACROSS_USERS_FULL	0.347
10	WRITE_MEDIA_STORAGE	0.320

It may seem surprising that the permission listed in Table II row 2, which allows apps to authenticate with your Google account, is the most indicative of benignity, but these requests require user interaction [25]. As such, they are not very useful to malicious app authors. The Bluetooth permissions, found in rows 3 and 5, may also seem odd at first sight. However, most malware infection happens over the Internet, not in person, so there are not many uses for Bluetooth communication for malicious app authors.

Table III
SYSTEM CALLS INDICATIVE OF MALICIOUSNESS

Rank	System Call	Weight
1	mkdirat	0.322
2	nanosleep	0.272
3	getdents64	0.149
4	fstatat64	0.138
5	clock_gettime	0.137
6	exit	0.130
7	pread64	0.124
8	getppid	0.112
9	fchmodat	0.108
10	futex	0.104

Of the maliciously classified system calls in Table III, probably the most logical one is `fchmodat`, found in row 9, because apps may want to set the executable bit on a script they're injecting, or make a file writable that isn't by default. [26] A steep drop off in the weights listed in column 3 was also noticed. This can be interpreted as there are only 2-3 system calls that are significant enough to be somewhat good indicators of maliciousness. The findings in Table III also match the results found in previous papers that showed a relatively low classification accuracy when using system calls.

Table IV
SYSTEM CALLS INDICATIVE OF BENIGNITY

Rank	System Call	Weight
1	readlinkat	0.621
2	renameat	0.333
3	faccessat	0.277
4	pwrite64	0.247
5	fsync	0.214
6	sendmsg	0.196
7	getsockname	0.161
8	gettimeofday	0.135
9	epoll_create1	0.130
10	pipe2	0.129

Given the relatively low classification accuracy of the system call data from previous papers, it's not worth spending much energy diving in-depth on every system call weight listed in Tables III and IV. However, it is noteworthy that `readlinkat` is by far the most heavily weighted system call. This system call only allows the app to read the contents of a symbolic link [27]. It is unclear how, if at all, this system call could be utilized in a malicious manner.

B. False Positive and Negative Rates

While classification accuracy is an important performance metric, it is not the only one. The rates of false positives and false negatives is also of interest. A low false negative rate is probably more important than a low false positive rate because it is better to be suspicious of some applications that turn out to be benign than to let malicious ones slip through the cracks and wreak havoc. To determine the false positive and negative rates, the TensorFlow script was modified to subtract the bit vector with the correct classifications from the vector with the predictions rather than just comparing them for equality. A -1 in the resulting vector represents a false positive and a 1 represents a false negative (0 corresponds to accurate classification). Percentages are then calculated for each. In Figure 1, the blue data points show the classification accuracy. The red data points are the false positive rate and the green ones are the false negative rate. Those two metrics are almost the same once the model has time to train properly.

IV. IMPROVEMENT ON PREVIOUS WORK

Previously, a classification accuracy of around 80% was achieved using permissions data and an accuracy of around 60% was achieved using system call data. One way these numbers can be improved is by tweaking the machine learning model.

A. API Calls

Thus far only permissions and system calls have been used as features in our machine learning algorithm, but there are many other features of Android applications that may be indicative of their maliciousness. One such feature is the code. While the unobfuscated source code is not usually available, much

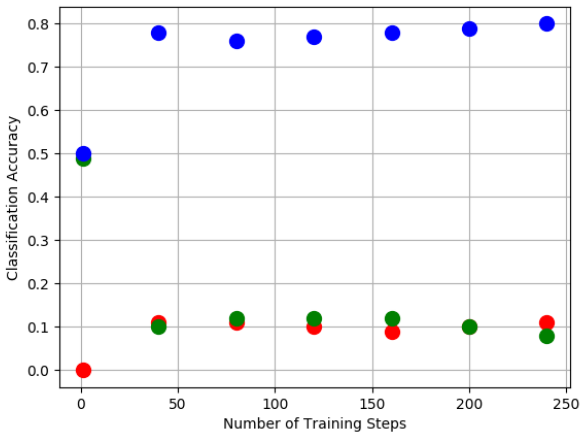

Figure 1. Number of training steps vs classification accuracy and false negative/positive rates.

can be learned from disassembled `jar` files. One feature that can be extracted from them is which methods were called in the Android and Java APIs. All apps share these two libraries.

In order to get the disassembled code from the .apk files, the following steps were executed:

1) Unpack the .apk file using the `unzip` utility
2) Convert the `classes.dex` file into `classes-dex2jar.jar` using the `d2j-dex2jar.sh` utility
3) Use a combination of the `jar` and `javap` utilities to print the disassembled code to a file:

```
$ javap -c -classpath
classes-dex2jar.jar $(jar -tf
classes-dex2jar.jar | grep "class$"
| sed s/
.class$//) > disassembled.code
```

The primary difficulty with getting this data into a usable format is that many app developers have used obfuscation techniques, either to prevent the copying of their source code or to prevent exactly this type of analysis. The obfuscation seems to work by having classes and methods with single letter names ('a', 'b', etc.) and calling the desired methods indirectly through those. Therefore, a simple heuristic was developed to prune such classes: any class with a name shorter than 4 characters or with a parent class only 1 character long was ignored.

If Android API classes (e.g. `android.app.Activity`) are included, the classification accuracy is 0.5, meaning nothing was learned from the data. This is probably due to the large size of the API (several thousand classes) relative to the code calling into it, leading to a lot of noise in the data.

However, by looking only at the Java API classes (e.g. `java.util.Random`), the model can successfully learn from the data. In Figure 2, the classification accuracies achieved using the same 200 app dataset that was previously used to compare system call and permission features are shown. As seen, the classification accuracy increased to around 75-80% as the model trained on the Java API call data using a 200 app dataset.

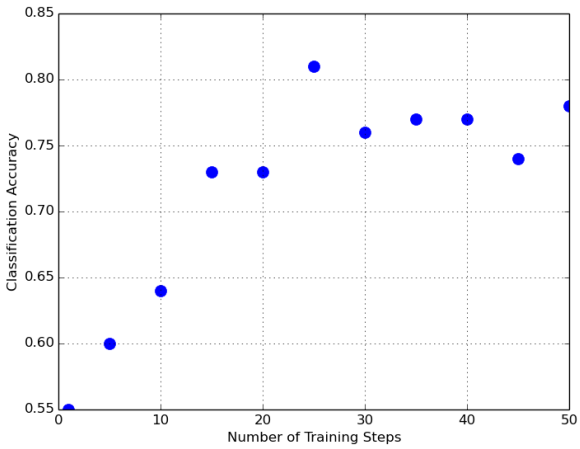

Figure 2. Number of training steps vs classification accuracy using Java API call data on 200 app dataset.

Java API calls are a more telling feature than system call data in determining maliciousness, and almost as accurate as permission requests. This is probably because the API calls give more specific information about what actions the apps are trying to perform, whereas many system calls can be used in a wide variety of contexts and for a wide variety of purposes.

While the initial experiment with Java APIs as the feature used the original 200 app dataset in order to get a fair comparison to the classification accuracies of system calls and permissions, it was also tested against a larger dataset. Figure 3 contains the results from an experiment using 2,000 apps (half benign, half malicious). As seen, the classification accuracy increased to around 82% using Java API call data on this larger dataset.

Tables V and VI list Java API classes that were found to be most indicative of maliciousness and benignity, respectively, for the 2000 app dataset experiment.

Table V
CLASSES INDICATIVE OF MALICIOUSNESS

Rank	Class	Weight
1	java.net.SocketException	0.309
2	java.lang.StringBuffer	0.303
3	java.lang.Character	0.294
4	javax.crypto.Cipher	0.282
5	java.io.ByteArrayInputStream	0.193
6	java.lang.UnsatisfiedLinkError	0.165
7	java.net.Proxy	0.161
8	java.util.Hashtable	0.161
9	java.util.zip.InflaterInputStream	0.157
10	java.util.GregorianCalendar	0.153

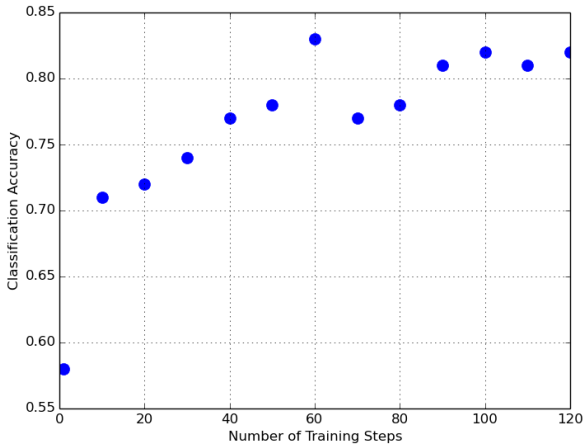

Figure 3. Number of training steps vs classification accuracy using Java API call data on 2000 app dataset.

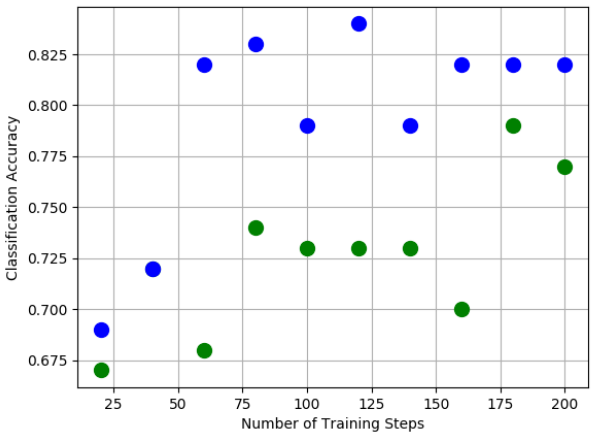

Figure 5. Number of training steps vs classification accuracy with and without learning rate decay.

Table VI
CLASSES INDICATIVE OF BENIGNITY

Rank	Class	Weight
1	java.util.AbstractList	0.266
2	java.util.concurrent.atomic.AtomicLong	0.229
3	java.lang.reflect.Constructor	0.218
4	java.security.Signature	0.209
5	java.io.FilterOutputStream	0.199
6	java.util.Currency	0.194
7	java.security.SecureRandom	0.188
8	java.lang.Throwable	0.177
9	java.nio.charset.Charset	0.174
10	java.lang.Runtime	0.173

Perhaps the fact that some of the classes most indicative of benignity, according to Table VI, are data structures is a reflection of the more careful coding practices used for benign applications. The presence of the security related classes, `java.security.Signature` and `java.security.SecureRandom`, is perhaps unsurprising since malicious app authors have little motivation to protect their victims from other attackers.

The larger dataset allowed the model to better learn how to distinguish malicious apps from benign ones. Not only this, but some of the features that were considered top indicators of maliciousness became considerably less significant with a larger dataset. For example, `Java.crypto.SecretKeyFactory`, which was the 5th most indicative of maliciousness in the 200 app dataset, became the 66th indicator of maliciousness in the 2000 app dataset. Since the small dataset can't be expected to be representative of all applications, such change should be expected.

B. Learning Rate Decay

In previous work, the learning rate was constant. This can be improved upon through exponential decay of the learning rate over the course of training the network. This prevents overfitting to the training data, and ultimately results in an increased classification accuracy. Figure 4 shows the exponential decay of a learning rate starting at 0.01 using the function $y = (0.01)(0.975)^x$ over the course of 50 training steps.

To compare the differences between a decayed learning rate and a constant learning rate, the 2,000 app dataset was tested using API calls as the feature for the machine learning algorithm with a learning rate of 0.01. Figure 5 shows the accuracy difference between a decayed learning rate, shown in blue, and a constant learning rate, shown in green, for every training step. As seen, the decayed learning rate helped improve classification accuracy by several percentage points. Without decay, the network is effectively overwriting what it learned early in the training cycle by overzealously updating the weights during later stages.

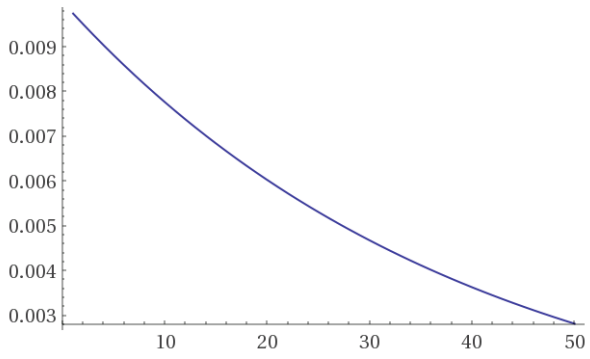

Figure 4. Exponentially decaying learning rate function from .01 over 0-50 steps.

C. Hyperparameter Optimization

In machine learning, a distinction can be made between parameters, such as the weights, and hyperparameters, such as the learning rate. While parameters are the values in the model that are affected by the training process, hyperparameters are the higher-level properties of the model that are chosen by the user [28]. Other examples of hyperparameters include the rate at which the learning rate decays, the number of training steps, and the size of sample chunks used for the training. Determining the best hyperparameter value is an important step. One method is to run the entire model multiple times using a range of values to determine what works best. However, the parameters can't be considered completely independently of each other because there are some combinations that can lead to undesirable performance. For example, combining a relatively high learning rate (greater than 0.01) and large chunks of samples (200 apps in each) for training led to instability in the algorithm that caused the weights to become `NaN`. This makes the model useless.

Figure 6 summarizes the data obtained that helped to determine the best rate of decay for the learning rate. In Figure 6, the classification accuracy increases as the decay rate increases to 0.98. Since the rate is raised to a power ($y = (0.01)(0.975)^x$), a higher rate actually means that the learning rate decreases more slowly over the course of the training. The reason the model isn't learning enough with lower (faster) decay rates is that the weights are only seriously affected by the first batches of samples, not later ones. Some outliers are to be expected since the process is non-deterministic to a degree. In Figure 6, a sample of 2,000 apps was used and the model was trained on the API methods called. The classification accuracy is an average of the ones achieved for 120, 160, and 200 training steps to reduce noise. The learning rate used was 0.005.

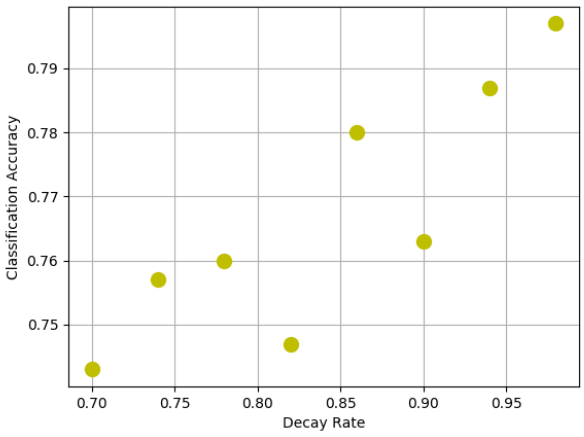

Figure 6. Decay rate of learning vs classification accuracy.

Figure 7 shows the average accuracy from training again on API calls using a decay rate of 0.98 and varying learning rates. The learning rate on the x-axis controls how much the weights in the network are updated on each run. The classification accuracy on the y-axis is an average for runs of 20, 40, 60, ..., 200 training steps. Since smaller learning rates take more training in order to achieve optimal accuracy, it wouldn't have been fair to compare them for a single number of training steps. As such, an average was taken instead. It can be conclude from Figure 7 that a learning rate of 0.0025 is optimal for this data and hyperparameters.

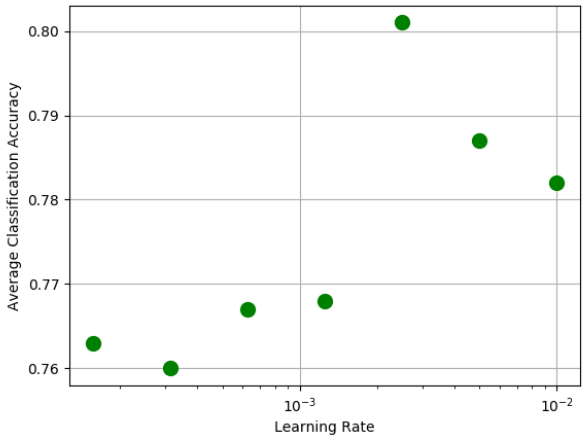

Figure 7. Learning rate vs average classification accuracy.

D. Making Experimentation Easier

In science it's useful to have an experiment that can run relatively quickly so that you can try different parameters and techniques to see what works within a reasonable amount of time. To achieve this goal, the software used for this work was modified to make it easier to run multiple trials, perhaps with different parameters. Both the results and parameters used are recorded automatically for later analysis. The software was also written in such a way as to make it very easy to use a different feature or a different machine learning model without having to change other parts of the data pipeline. Since this may be of use to other researchers, it's open source here: https://github.com/mwleeds/android-malware-analysis.

V. Conclusion

Mobile malware is a constant threat for Android users. As these devices become increasingly important in our daily lives, it is of the utmost importance to ensure their safety and security. As such, the development and testing of new, effective and efficient malware detection techniques must be a priority. In this research, the use of Java API call data was examined and compared to the previously tested permissions data and system call data features. The specific permissions, system calls, and Java API calls that proved to be the most

indicative of maliciousness or benignity, according to the machine learning model, were also found. Furthermore, the impact on classification accuracy of using a decaying learning rate, of hyperparameter optimization, and of randomized validation was examined, and a considerable improvement in our classification accuracy through these techniques was achieved. Java API call data proves to be helpful in the classification process. Ultimately, an 82% classification accuracy with Java API call data to distinguish between a malicious and benign app was achieved.

VI. Future Work

The utilization of different machine learning models, such as Support Vector Machines, is our main priority for future work. We also plan to combine all of the features into one model to further improve classification accuracy. Finally, we will strive to add more features to our model, either through static or dynamic analysis of applications.

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] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[20] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro, "Droidscribe: Classifying android malware based on runtime behavior," in *Security and Privacy Workshops (SPW), 2016 IEEE*. IEEE, 2016, pp. 252–261.

[21] 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.

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

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

[24] M. K. Matthew Leeds and T. Atkison, "A comparison of features for android malware detection," in *Proceedings of the 2017 ACM Southeast Regional Conference*. ACM, 2017.

[25] I. Google. Android developer documentation: Manifest.permission. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission.html

[26] Linux. Linux man pages: fchmodat. [Online]. Available: http://man7.org/linux/man-pages/man2/fchmodat.2.html

[27] Linux. Linux man pages: readlinkat. [Online]. Available: http://man7.org/linux/man-pages/man2/readlinkat.2.html

[28] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.