Detecting Malicious Android Applications Utilizing Neural Networks

Aleksandr Kovalev
Tzipora Halevi
alex.kovalev010@gmail.com
halevi@sci.brooklyn.cuny.edu
Brooklyn College, The City University of New York
Brooklyn, New York

ABSTRACT

The Android application market grows rapidly, benign and malicious applications constantly change. This work looks into detecting malicious Android application utilizing different neural networks, and proposes a newly designed customized neural network for Android classification. To test this algorithm, we utilize the CICAndMal2017 dataset [5]. We further compare our algorithm to the existing Drebin SVM-based algorithm [1]. Both algorithms use features extracted from the Android .apk files, such as hardware components, permissions requested, app components, filtered inter-processes, restricted API calls, used permissions, suspicious API calls, and Network addresses.

This study introduces a newly designed customized neural network written in Python, using python's powerful libraries in machine learning. The neural network is optimized, trained and tested on the CICAndMal2017 dataset, which includes android apps dating between 2015-2018. In parallel, to test the Drebin's algorithm longevity on newer data, we train and test it on this new dataset. This work introduces the performance results for both tests, and examine the strengths and weaknesses of both methods of classification.

ACM Reference Format:
Aleksandr Kovalev and Tzipora Halevi. 2019. Detecting Malicious Android Applications Utilizing Neural Networks. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Statistics show that Google Android continues to be the most popular mobile OS in the world. As a result, the Android application market grows rapidly, with 60% growth in number of apps downloaded globally between 2015 and 2017. [**?**] Google allows users to install applications from unofficial sources, i.e., ones that have not been vetted by Google. This in turn provides a way for malware to attack mobile phones. Therefore, as malware infection rate of smartphones continues to soar, Android devices have been shown to often be their target, as 81% of all infections are estimated to attack Android systems [**?**]. newly introduced malware samples have still been estimated at 2.68 million in 2018 [**?**], showing the threat is still significant to users. The detection continues to be challenging as malware applications constantly change and adapt to evade detection.

Researchers have been looking into solving this problem utilizing different methods. While dynamic monitoring can be highly effective in detecting suspicious activity, they also present challenges, such as significant runtime-overhead. On the other hand, static analysis methods, such as Drebin [**?**] abd RiskRanker (Grace et al. 2012, citeGrace12), introduce smaller run-time overhead.

To protect the privacy of the user and the device security, Android provides the permission system. Applications must request permission to access certain system features/hardware components, such as camera and internet, as well as private user data, i.e., contacts and SMS history. The system may grant a permission automatically or prompt the user for approval, depending on the feature type. However, studies show that most users do not pay attention to the permissions requested, and even those who do may not understand them correctly [**?**]. As a result, users tend to blindly grant permissions, thereby undermining the goal of the permission system. and limiting its potential effectiveness.

Our work looks into detecting malicious Android applications utilizing machine learning algorithms. We utilize the CICAndMal2017 dataset, which includes over 5,000 (426 malware and 5,065 benign) of samples collected on real devices on real devices.

2 CONTRIBUTION

This paper proposes a newly designed customized neural network that is trained and tested on a real-world Android application dataset. This work utilizes the Keras library (released March 2015) a high level API built on Google TensorFlow (TF). Keras is considered more user-friendly and easy to use as compared to TF, and enables rapid prototyping of both simple and complex neural networks[**?**]. This proposed neural network is trained and tested on a recent mobile malware dataset. allowing for the system evaluation

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

on a set of recent benign and malware Android applications. The network is optimized and tested using different parameters, and our results show that the algorithm achieves good performance using different evaluation metrics.

In addition, we test and compare the results of our algorithm to a previously proposed SVM-based Drebin malware detection algorithm, which is based on the scikit-learn [6] (released in 2007). The Drebin original algorithm is trained and tested on the same new dataset, which provides two benefits: testing the performance and longevity of this algorithm on a dataset which includes new malware applications, and comparing its results on the same dataset to the newly designed neural network. Our study shows that even though malware continuously evolves, the Drebin algorithm (2012) performance on the new dataset is still comparable to the original findings. In addition, our results show that while for certain thresholds our neural network provides comparable results to Drebin, it is possible to adjust the parameters and achieve improved recall , which is an important parameter for malware detection, utilizing the newly proposed NN.

3 DATA VECTORIZATION

The CICAndMal2017 dataset is analyzed utilizing functions from the Drebin software. This is done by extracting features from the .APK files. We then place the features within a table that is then used as input to the neural network.

To create a vector space of the features, we utilize the functionality of the Drebin software. The functions used scans each individual .apk file, and the values extracted from the .apk files are combined and placed within a vector space. The vector space is used to train a support vector machine model (SVM) using Pythons sklearn library. The library model used for the random classification problem is GridSearchCV. The values extracted from the .apk files are combined and placed within a vector space. The SVM then calculates the hyperplane that separates both classes. This acts as a classifier and separates the malicious from the benign applications.

To convert the text data Drebin extracted into vectors for our neural network we used CountVectorizer from the scikit-learn library in Python which handles the conversion. The challenge was to extract the permission request without breaking them apart into multiple words. The default configuration within CountVectorizer separates words by character symbols that represent dividers within the language. For example, "android.permission.INTERNET" would be separated into 3 words; android, permission, and INTERNET. Because the "." is considered the divider. This presents a problem for us if we want to maintain the syntax of the permission requests. To solve this we had to create a new regex which would replace the default token pattern in the vectorizer function. The regex format used is: [u'(?u)"(.*?)"']. This Regex separates only the strings within the quotations and keeps the original text unchanged. Insuring that a proper set of words would be transformed for each application. The a transformation example can be seen in Figure **??**.

centering

```
["android.permission.INTERNET", "android.permission.WRITE_CONTACTS"]

                        -into-

android.permission.INTERNET, android.permission.WRITE_CONTACTS
```

Figure 1: Example of Text With Our Regex Format Applied

4 NEURAL NETWORK ARCHITECTURE

The dataset for the neural network is placed in text form within a CSV file. To feed the data to the network we had to change the text data into numerical data to allow the network to work with the data. The methodology used for our transformation of text to numeric is the 'bag of words' model, which transfers text into a word multiset used as the neural network input data. For example, each word can be likened to a marble contained in a bag. The bag has a collection marbles (words) which represent the feature set for an application. Each bag can be compared to another bag by reviewing the marbles that are or are not in each bag. In our dataset each word within a feature set is given a value in a vector. The size of the vector corresponds with all possible words within all of the sets with in a category. If the word is used the location of the word in the vector is marked with a one. The words within the dataset are mostly predetermined system calls which are standardized by the Android operating system. The words follow a specific syntax to operate within the android environment. This means that each feature request with in an application is identical to other applications that operate within Android. For Example, 'android.permission.SEND_SMS' is used by any application that needs to send SMS. By labeling the location of 'android.permission.SEND_SMS' within a vector containing all used permission requests the neural network can easily distinguish between applications. The neural network can then learn which combination of words (permission requests) are associated to malicious applications. Benign or malicious applications should show patterns within the vectors which could be detected using machine learning algorithms. Here is an example of a list of one applications permission requests within the RequestedPermissionList:

["android.permission.CHANGE_NETWORK_STATE",
"android.permission.SEND_SMS",
"android.permission.RECEIVE_BOOT_COMPLETED",
"android.permission.READ_PHONE_STATE",
"com.android.alarm.permission.SET_ALARM",
"android.permission.SYSTEM_ALERT_WINDOW",
"android.permission.ACCESS_WIFI_STATE",
"android.permission.WRITE_SMS",
"android.permission.ACCESS_NETWORK_STATE",
"android.permission.WAKE_LOCK",
"android.permission.GET_TASKS",
"android.permission.CHANGE_WIFI_STATE",

"android.permission.RECEIVE_SMS",
"android.permission.READ_CONTACTS"]

The above Android permission requests and other system calls follow protocol. This makes the text standardized and each one can be turned into a unique vector that represents this unique set of requests. Each category of data which is analyzed will contain its own vector space. This allows us to create a table were each cell contains a vector that represents the application. Over multiple categories each application will have a set of vectors which belongs to itself. Finding the relationship between the sets of vectors that represent malicious or benign applications will be the neural networks job.

4.1 Network Configuration

After we prepared the data, we began the architecture creation process for the neural network. Our goal was to create a neural network that would perform very well at detecting malicious applications. This leads us to focus on reducing the number of false negatives towards malicious applications as much as possible. In the spirit of Drebin's paper we also wanted to replicate there constraint on false positives as much as possible. "We choose a false-positive rate of 1% for DREBIN which we think is sufficiently low for practical operation" (Drebin). After experimenting with multiple configuration settings within Keras's sequential network we settled on two architectures. We created two neural networks with separate objectives that we felt are important. The first neural network (NN1) prioritizes the detection of malicious applications over any other parameter. The second neural network (NN2) focuses on adhering to Drebins benchmark of 1% false positives as close as possible.

Both models are created using Keras's sequential model. Keras models are trained on Numpy arrays of input data and labels. All backend functions for the network are handled by Tensorflow. "Keras is a model-level library, providing high-level building blocks for developing deep learning models. It does not handle low-level operations such as tensor products, convolutions and so on itself" (Keras site). All the data that is input into the network is converted into tensors, which is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes (3). All mathematical functions within the network are done onto and with tensors. Keras provides us with easy-to-use features and functions which modify the neural network to our needs.

4.2 Challenges

The greatest challenge we faced was configuring the neural network to work with an imbalance in the data. The dataset contains a large number of benign applications and a relatively small number of malicious applications to work with. While testing several configuration options provided by Keras we discovered a few patterns regarding which configuration settings had the most effect on the results. The most impactful configuration setting we observed is the class weight parameter. It was a challenge finding the correct class weight parameters to aid the network in its learning. Keras's default settings for their neural network does not require custom class weights. Using the default setting our neural network did not perform well. We tried a setting to allow Keras to decide the class weights for us by setting class_weight to 'Auto' & 'balanced'. This setting had a negative effect on our tests. Detection performance dropped with one test run failing to detect a single malicious application. We decided to customize the class weights ourselves to improve performance over the default settings. After many trials we found a weight range that worked best for training our neural network. The class weight range we settled on is a weight 0.5 for benign and 4 to 8 for malicious. This gave us a weight ratio range from 1:16 to 1:8 of benign towards malicious. We discovered that moving the weight of malicious up or down had predictable results during our training and testing of the network. Setting the class weight of malicious to 8 gave us the best detection rate overall. As we incrementally lowered the setting we noticed that the detection rate lowered slightly while improving rates in other areas, like reducing our false positive rating for benign applications. After further trails, we determined the lowest weight that provided the best balance between malicious detection rate and false positive benign rate was 4. Anything under 4 degraded the effectiveness of our detection model too greatly to consider.

Setting the class weights this way creates an artificial balance to the training data for the neural network. During the learning process, the network favors mistakes made towards malicious applications more so than mistakes made towards benign. This creates a training environment that focuses more on the imbalanced data of malicious application.

The next configuration settings that had the most impact on our results were; the outer layer activation function, the number of nodes within our hidden layer, and the number of epochs. These configurations worked together to improve our results. All 3 of these configurations had a delicate balance amongst each other. Changing one setting we found would require changing the others to find an optimal balance. For example, as we added more nodes we noticed that the network needed more time to train. So we had to increase the number of epochs. Adding more epochs showed negative effects with certain activation functions and we had to change them as well. Class weights also had an effect on these changes. We noticed that when we increased the class weight of malicious to improve detection rate we had to reduce the epoch count because we witnessed a decrease in overall network performance. To find the best balance, fine tuning these configuration options was a constant task. Throughout or tuning of the neural network we discovered two model configurations that showed good results in two different ways. One model configuration showed the best malicious detection rate and another model configuration was the best compromise between detection rate and false positive rate. We separated both models and designated them NN1 and NN2. NN1 focus

is malicious application detection and NN2 is focused on reducing benign false positives.

4.3 Neural Network 1

Our focus for NN1 was guided by the importance of minimizing the amount of malicious applications that escape detection because it only takes one malicious application to inflict serious damage. Configuring the network to focus on malicious detection does effect the false positive rate. As the network gets better at detecting malicious applications it will mislabel some benign applications. We considered this as an acceptable loss as long as the false positive rate is within reason and manageable.

The best model we found for this purpose was a network with one dense hidden layer containing 200 nodes. NN1 structure is one input layer, followed by the hidden layer, and ends with a one node output. The input layer consisted of nodes equal to the number of input parameters given. After the data pass through the input layer it moves to the 200 nodes in the hidden layer. The input layer and the hidden layer are densely connected, each node is given a connection to each other node in the next layer. All 200 nodes are then connected to one output node which is the classifier node. The output node only outputs a binary value of 0 or 1. The activation function for the hidden layer is 'relu', which stands for 'Rectified Linear Unit'. The activation function for the output node is 'hard_sigmoid'. 'hard_sigmoid' is segment-wise linear approximation of sigmoid. This combination of layers and activation function we found works best with our dataset.

The loss function was set as 'binary_crossentropy'. The optimizer for the learning stage of the NN1 selected is 'SGD', which is Stochastic gradient descent. This optimizer showed the best results for our network. It reduced the learning rate of the network which improved malicious detection rate. Other optimizers made the learning rate to fast had lower accuracy scores overall with our dataset. We noticed that smaller increments of learning rate improved overall performance. The number of epochs used within NN1 is 50. This setting is the number of complete passes that the network does for training. This in combination with the SGD optimizer creates a stead learning curve with our dataset and showed the best results. The batch size we used is 32. This is the number of training samples that are processed through the network at a time. After 32 samples have been processed the network models parameters are then updated.

To improve malicious detection with NN1 we had to deal with the issue of imbalanced data. The population samples within our dataset is highly imbalanced. An imbalance of data can lead to a neural network that favors the most popular sample for its learning process. This can lead to misleading results. As the accuracy can be high for the overall dataset test but all the samples that are small in number are wrongly classified. This is due to the network over training itself on the most frequent sample while neglecting the less frequent samples. Our population of 6% malicious applications created a major imbalance. NN1 without proper tuning was naturally learning and favoring the benign applications. This was countered by setting the class weights to a ratio of 16:1 (malicious towards benign). This is that same ratio of the benign to malicious application population within our dataset. The ratio we used for the class weight parameter that we found worked best was 0.5:8. This created an artificial balance to the training data for NN1. This class weight modify the weight of mistakes that the network makes during training. This setting makes the network favor mistakes made towards malicious applications more so than mistakes made towards benign application. This creates a training environment that focuses more correctly labeling malicious application.

4.4 Neural Network 2

NN2 was configured to adhere to Drebin's claimed 1% benign false positive rating as a benchmark. Some settings carried over from NN1. We changed the following settings to reduce benign false positive rating as much as possible. After testing many configurations we are left with the following configuration settings. This model contains 1 hidden layer with increased node count of 1,000, The activation function for this layer was kept as 'relu'. The output layers activation function was changed to 'sigmoid'. The loss function was kept as 'binary_crossentropy'. The number of epochs was increased to 100 to allow the network more time to learn with the increased network size. We noticed that for this model configuration during long epochs 'hard_sigmoid' created fluctuations in accuracy during training and validation. Setting the outer layer activation function to 'sigmoid' smoothed out the learning process of our neural network. The optimizer was kept at 'SGD' and the batch size is unchanged at 32. The class weight was adjusted to offset some of the favoritism towards malicious applications and allow benign applications more weight. The class weight was set to a ratio of 1:12 (benign towards malicious) from 1:16 in NN1. This was done to allow the network to learn benign applications better without sacrificing to much on malicious application detection.

There is an additional model(NN3) which has the class weight ratio lowered even further to 1:8 and uses 'hard_sigmoid' as the activation function for the outer layer, all other settings are identical to NN2. NN3 achieves a slightly better false positive rating then NN2. However the overall performance over multiple runs of NN3 was slightly worse then NN2. We suspect his is due to added noise during training and validation of the network which is caused by 'hard_sigmoid' as the outer activation function. The results and further explanation of NN3 is located in the 'Results' section further down.

4.5 Metrics

Both neural network models used scikit-learn library metrics to present the same metrics that Drebin presented. Keras does not provide these metrics by default, we had to add them using the scikit-learn library see Figure **??**. These additional metrics allow us to see how the networks are actually performing on the given dataset and allows us to compare them to Drebin's results. Since our dataset is highly imbalanced these

metrics provide us insight on our models true performance to given categories. In the image, 'support' stands for the number of samples within the test dataset. The most important focus for NN1 is the recall score for the malicious row. Recall is an important metric that measures how successful the model actually is at identifying real malicious applications. A low score signifies that the model fails to detect some malicious applications. A high score signifies that the model detects all or nearly all malicious applications. Our metrics of focus for NN2 is both recall scores for benign and malicious. The recall score for the benign category represents the number of benign application properly labeled. Any benign applications labeled as malicious would be considered a false alarms and would lower the recall score. NN2's goal is to maintain a low false positive rate of benign applications while trying to maintain as high of a recall score for malicious as possible. A recall score of 1.00 signifies that no benign applications were mislabeled as malicious.

	precision	recall	f1-score	support
Benign	0.99	0.96	0.98	104
Malicous	0.87	0.96	0.92	28
micro avg	0.96	0.96	0.96	132
macro avg	0.93	0.96	0.95	132
weighted avg	0.96	0.96	0.96	132

Figure 2: Neural Network Additional Metrics To Better Understand The Networks True Performance

5 PRE-PROCESSING

In order to utilize the vectorization functions available in Drebin, certain implementation changes were needed. Testing the original Drebin code showed that the code was not operational at its current state found online, which may be due to the fact it was developed a few years earlier (2010 - 2012). In particular, many library names have been renamed over the years. We had to modify one line within the RandomClassification.py file to get the program to run. The following library was renamed, 'from sklearn.cross_validation import train_test_split' to 'from sklearn.model_selection import train_test_split'. This was done because cross_validation has been renamed to model_selection in the scikit-learn library. Drebin was given the current dataset which was newer then its own dataset. The program functioned and produced vectorized space data. However, during training and testing there were convergence warnings and f1 score calculation errors given by scikit-learn library. The training was deemed sub optimal for the current dataset and some parameter adjustments were made. We adjusted the C penalty parameters for the GridSearchCV model within the RandomClassification.py file. The original parameter setting used was [0.001, 0.01, 0.1, 1, 10, 100, 10000], this was changed to [0.1, 1, 10]. This parameter setting removed all warnings and errors from training and testing and improved the overall performance of Drebin's classifier.

The neural network we created was configured to use the same features as the original Drebin software. These features were vectorized into a Comma-separated values (CSV) file. In order to create a CSV file we added a function within Drebins code in the GetApkData.py file. The function made a copy of all data that Drebin scanned and extracting from the files and placed it within a table. The function copies the data stored within the variables inside Drebins code (without interfering with any other functionality). These same variables are used to create the .data files which Drebin uses for the classifier. This allowed us to create a CSV file of all the individual .data files Drebin creates.

Run-time details: Drebin operates by scanning android applications contained within .apk files. The applications were separated within two folders. A folder containing good .apk files and a folder containing bad (malicious) .apk files. Using built in python libraries, Drebin scans each file and extracts data. The data is stored in individual data files and linked to each Android application. Drebin uses this data for training and testing its classifier. When Drebin starts for the first time it scans the good and bad folder for any new .apk files. Drebin will scan each .apk file that does not have its features already extracted. This leads to the initially run of Drebin to be slow, because it scans each file to prepare the data for its model training. This process can take several hours. But once the data is extracted from the .apk files the program only takes seconds to complete. Since it only reads the data files containing the features that Drebin needs for its classifier.

6 TEST PARAMETERS AND DATASET

The dataset that we chose to use for training and testing was the CICAndMal2017 dataset [**?** **?**]. It includes a collection of Android applications (dating 2015-2017). The dataset contains benign and malicious applications .APK files. The malicious applications come from three categorizes: Ransomware, Scareware, and SMS Malware. Each category is further divided into unique malware families. We used 100 malicious application evenly and randomly selected from each category and their families. The number of benign applications used is 1,600, resulting in 6% malicious application percentage. This was comparable to the malware percentage in the original Drebin dataset (which was about 5%). The malicious application pool did not contain applications that are from the adware category from the CICAndMal2017 dataset. This is inline with the original Drebin work, which removed those applications, as they stated that 'this type of software is in a twilight zone between malware and benign functionality' [**?**]. We also used a 66% training and a 33% testing split, similar to that research.

Testing against the Drebin dataset: Drebin's official site only provided access to the bad .apk files that the original authors worked on, which are from 2010-2012. They did not offer access to the original benign application .apk files (which the CICAndMal 2017 dataset did include). We did not use the newer benign application from the CICAndMal 2017 with

the older malicious application from Drebin. We felt that this would tarnish that dataset as this is not a realistic scenario.

As malware continuously evolve to evade detection, and in addition, since a recent work found duplication issues in Drebin [**?**], using a newer dataset gave us an opportunity to test the longevity and performance of Drebin's detection method on this newer dataset.

6.1 Features Set

Both the neural network and Drebin were trained and tested on the same dataset. The input files were vectorized using the Drebin software and an input file was created as input to the the newly designed Neural Network. The dataset includes a collection of Android application feature requests and behaviors. The features analyzed are as follows:

- RequestedPermissionList: Permissions granted by the user at installation, allowing an application to access certain resources on the phone.
- ActivityList, ServiceList, ContentProviderList, BroadcastReceiverList: these components are declared by the application in the Android manifest. As some malware use known specific components, these are added to the dataset.
- HardwareComponentsList: Set of hardware components the application is requesting permission to access.
- IntentFilterList: abstract description of an operation to be performed. Inter-process and intra-process communication on Android is mainly performed through intents, by using passive data structures exchanged as asynchronous messages. Intents also allow sharing events information between different applications.
- UsedPermissionsList: Permissions that were actually used by the application. This may be a subset of the permissions which were requested, as some permissions may not be used during run-time.
- RestrictedApiList: Restricted API list used by the application for which permissions have not been requested.
- URLDomainList: Network connections established by the applications. While also used by benign applications, malware may regularly establish network connections to retrieve commands or transmit data collected from the device. Some of these addresses may appear in different malware samples, as they may involve botnets, and therefore may be used for classifications.

6.2 CSV File

The CSV file is comma delimited and has the following layout: The first column is the label, it consists of binary values that represent the application within the row (each row represents one Android application). A label of 1 value is given to applications that are malicious and a 0 value to benign. The other columns of the table represent the feature fields as follows: RequestedPermissionList, ActivityList, ServiceList, ContentProviderList, BroadcastReceiverList, HardwareComponentsList, IntentFilterList, UsedPermissionsList, RestrictedApiList, URLDomainList. Each section within the table has a text string of data that represents the column and the application it is from. Since there are comma separated strings within the feature sets themselves, we used brackets ([]) to keep the data intact.

Note: Drebin failed to extract any data on some of the .apk files given. We found only 10 files had no data, removed them from the CSV file and from the folder directories that Drebin trains with. Since these were blank data fields we considered them useless as nothing can be learned from them for a classification model.

6.3 Test Split

We ran our customized neural network 30 times and recorded the metric data. To calculate the effectiveness of malicious application detection we calculate the mean and standard deviation of each program for all 30 runs. The dataset was shuffled and randomly split for each run. The data split was maintained at 66% training and 33% testing. 30 iterations is a substantial amount to see where each program stands. This procedure was then repeated for the Drebin software, in order to test and compare the results.

7 RESULTS

To test the performance of each program, it was run on the full dataset, choosing in random 66% of the data for testing and 33% for training. Each program was cycled 30 times. The metric we focused on is recall. Recall is an important metric that can measures how successful the model actually is at identifying real malicious applications. Recall is a percentage that measure how well a model actually discovers a particular class of data. In our instance this would be how well the model detects actual malicious application. The calculation for recall is:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

The goal in any detection model is to decrease the number of false negatives. The more malicious applications that escape detection the greater the risk to the system. Each program outputs the same metric format which makes them easy to compare.

7.1 Neural Network 1

NN1 performed very well at detecting malicious applications. With 30 runs on a randomly split dataset, we achieved a mean of 90.2% for the recall score. The standard deviation of NN1 malicious recall is 3.6%. The individual results and the bell curve for NN1 are include in Figure **??** & **??** on page **??**. The highest recall score achieved by NN1 is 96%

and the lowest score given was 81%. NN1 on average missed 9.8% of the malicious applications within our dataset. This neural network was configured to maximize the recall score for malicious applications. From all 30 runs of NN1, all recall scores were above 81% and half of the recall scores are above 90%. To achieve these numbers the average benign false positive score dropped to 3.7%.

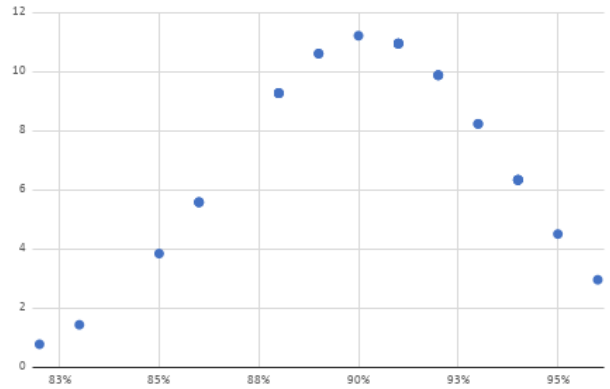

Figure 3: Neural Network One Confriguration Bell Curve Distribution

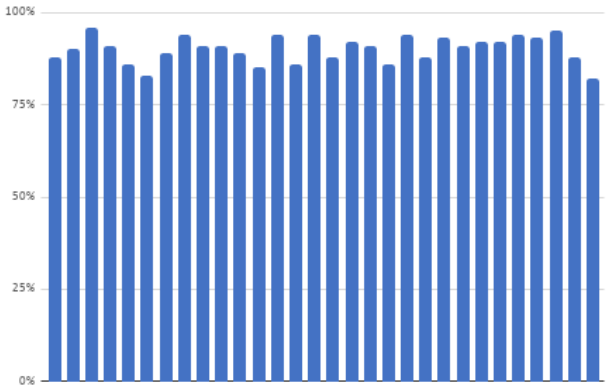

Figure 4: 30 Test Runs of Neural Network One Confriguration

7.2 Neural Network 2

For NN2 we configured our neural network to as close as we can to achieve a low benign false positive score of 1% (similar to the results in [**?**]).

We have reconfigured our neural network to see what we can achieve at the same benign accuracy score range. This was rather challenging. As we modified our neural network to better classify benign applications correctly we suffered on classifying malicious applications correctly. NN2 is the result of our configuration. The comfortable compromise we achieved was an average benign false positive rating of 1.6%, while maintaining a rating of 87.3% recall towards malicious applications. The standard deviation of NN2 malicious recall is 4.0%. The highest recall score achieved by NN2 is 94% and the lowest score given was 79%. NN2 on average missed 12.7% of the malicious applications within our dataset. This neural network was configured to maximize the recall score for malicious applications while maintaining a low benign false positive rating.

We want to note that in our testing we discovered a configuration(NN3) that achieved a false positive score of 1.2% towards benign applications by adjusting the class weight ratio to a 1:8 ratio and setting the outer layer activation function to 'hard_sigmoid'. The average score for recall for this configuration was 87.7%. Slightly higher then NN2. However the standard deviation suffered and we observed more recall scores that fell below 80%. We choose to keep and support the configuration of NN2 because we considered the results over all 30 runs to be more consistent. Results for NN2 are in Figure **??** & **??** on page **??**.

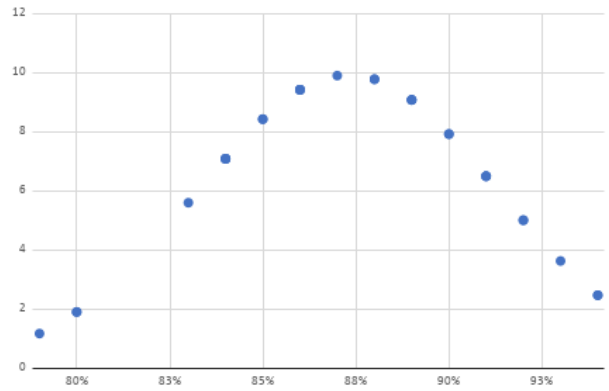

Figure 5: Neural Network Two Confriguration Bell Curve Distribution

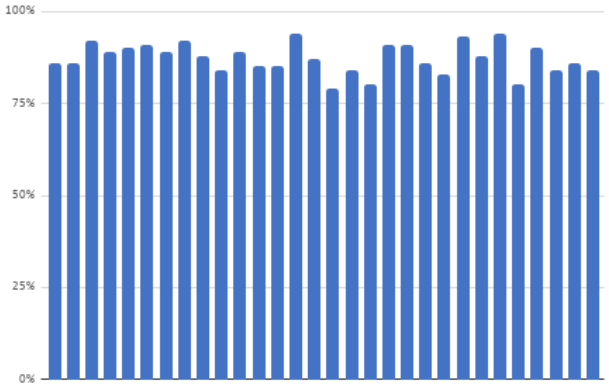

Figure 6: 30 Test Runs of Neural Network Two Confriguration

7.3 Comparison to the Drebin algorithm

Compared to results from the original Drebin paper, Drebin underperformed. Out of 30 run of the program on randomly split dataset we achieved a mean of 86% for the recall score. However, the false positive rating maintained the same as the paper. Drebin maintained a percentage of 1% mislabeled benign application. The original paper stated that, 'On the full dataset DREBIN provides..detection of 93.9%'(Drebin paper). This was calculated from a mean of 10 runs on their full dataset. From our dataset and 30 runs, Drebin achieved an average recall score of 86% on the dataset with a standard deviation is 6.3%. Charts containing the test results of Drebin are include in Figure **??** & **??** on page **??**. The highest recall score achieved by Drebin is 97% and the lowest score given was 69%. Drebin on average missed 14% of the malicious applications within our dataset. During the 30 test runs we observed a false positive rating of 1.x%.

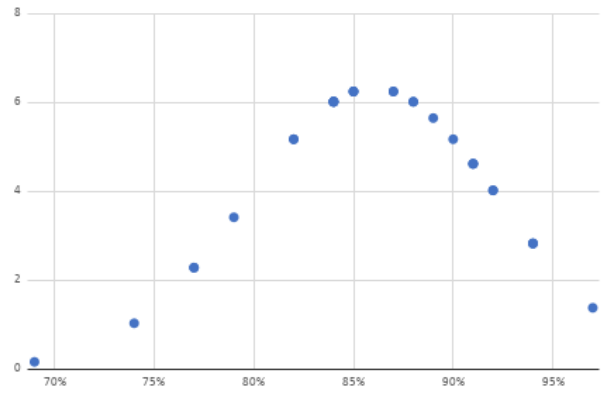

Figure 7: DREBIN Bell Curve Distribution of 30 Runs

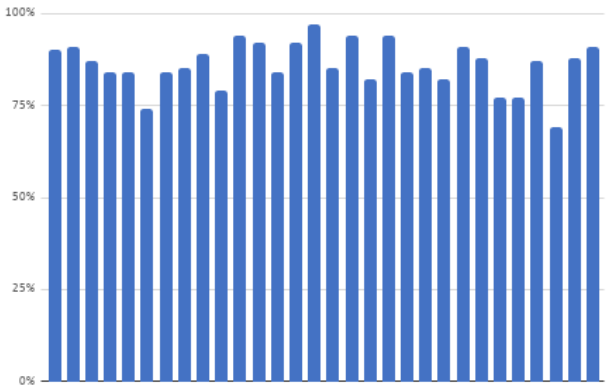

Figure 8: List of Results Containing 30 Runs of DREBIN

8 DISCUSSION

The challenge faced with training our neural network to detect malicious applications was the imbalance of the data. This imbalance was instilled intentionally to mimic the real world environment. The neural network without proper tuning was naturally learning and favoring benign applications. This was counteracted by setting the class weights to a ratio of 1:16 (benign towards malicious) for NN1. This created an artificial balance to the training dataset for the neural network. This aided the learning process and improved malicious application detection. NN1 performed very well at detecting malicious applications. NN1 only missed 9.8% of malicious application in the test set on average. The cost for this effectiveness was a benign false positive score of 3.7%. We feel that this score is acceptable in a practical sense. The detection of malicious applications is top priority. According to App Annie 2017 Retrospective Report, the average smartphone user has 80 applications on their phone and uses 40 of them in a given month [**?**] . Using these statistics would mean that 3 benign applications would be labeled as false positive if the user downloaded 80 applications. In a practical sense these false positives can be warnings that pop up notifying the user that the application downloaded may be malicious. The user can do further research regarding the application or ignore the warning if they trust the developer.

In the spirit of the original paper from Drebin we created a model(NN2) with a false positive rating of 1.6% towards benign applications. Out of the average of 80 applications on a users smartphone, this would be about 1 application on average was falsely labeled. With these average usage statistics, NN2 achieves the same result as Drebin does with benign classification. NN2 slightly outperformed Drebin with regard to malicious application detection. Drebin on average achieved a recall score of 86% and NN2 achieved 87.3%. However NN2's standard deviation was 2.3% lower than Drebin's. Drebin had a standard deviation of 6.3% while NN2 has 4%. Drebin has 5 test runs that resulted in recall scores below 80% while NN2 has only 1 test run. NN2 appears more consistent on the dataset we used. The results from Drebin are still impressive considering that it was created in 2012.

Throughout our testing we noticed that certain splits of data for training and testing presented challenges for NN2 regardless of configuration. We tried to double the node numbers to 2,000, added more hidden layers, adjusted class weights further, and various other tweaks. The results for the most part were the same. For these challenging splits of data the recall score for malicious application stayed within the 70's percentile. No matter the configuration used we could not achieve a recall score above 80%. We believe that this is due to the number of applications we have in our dataset. Perhaps with more data we could overcome this challenge.