Detecting Malicious Android Applications Utilizing Neural Networks

Aleksandr Kovalev, Tzipora Halevi

Brooklyn College (CUNY)

# Abstract

Our work looks into detecting malicious Android application utilizing different neural networks. To this end, we use the CICAndMal2017 dataset created by Canadian Institute for Cybersecurity. We use Drebin, shared in the “DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket” paper. In the original work, predictions were made based on an SVM algorithm as their classifier, by using different features extracted from the Android .apk files. The feature set examined includes Hardware components, permissions requested, app components, filtered inter-processes, restricted API calls, used permissions, suspicious API calls, and Network addresses.

The android application market grows rapidly, benign and malicious applications constantly change. We propose that a neural network may be the better classifier algorithm. The original research was created around android applications that are dated between 2010-2012. We design a customized neural network, which is written in Python, using python’s powerful libraries in machine learning. The neural network is trained and tested on the CICAndMal2017 dataset. In parallel, to test Drebin’s longevity on newer data, we ran the original SVM algorithm on android applications dating from 2015-2018.

To assess the effectiveness of both algorithms in the current android market state, we further test and compare the performance of both the newly designed algorithm and the original Drebin algorithm on a dataset of recent Android applications. We compare the results and examine which algorithm is more successful in identifying malicious android applications. We also examine the strengths and weakness of both methods of classification.

# Introduction

Our work looks into detecting malicious Android applications utilizing machine learning algorithms. To this end, we use the CICAndMal2017 dataset created by Canadian Institute for Cybersecurity. We use Drebin, shared in the “DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket” paper. In the original work (DREBIN), predictions were made based on an SVM algorithm. We propose that a neural network may be the better classifier algorithm. We created a neural network using Python’s powerful libraries in machine learning. The neural network and DREBIN are trained and tested on the same dataset. We then compare the results and examine which algorithm is more successful in identifying malicious Android applications.

# Contribution

We are conducting this research to see how well Drebin has aged within the current Android application market. The Android Application market is changing and growing rapidly. “The numbers from 2017 certainly inspire confidence. There’s been a 60% growth in the number of app downloads globally since 2015” (4). Drebin was created in 2012 and the marketplace has grown rapidly ever since. With a new Android marketplace we propose new technology to tackle the challenges presented with such a rapidly growing industry. The Keras library was initially released in March 2015 and Google Tensorflow was released in November 2015. These technologies are relatively young compared to scikit-learn which Drebin model is based on. scikit-learn was initially released in June 2007. Neural networks have increasingly become a popular tool to tackle problems and challenges that old algorithms tackled. We will be testing how the latest neural network technology performs against an older proven method.

# Classification Models

Drebin is the program with two functions. The first function scans each individual .apk file, extracting any information it needs for its second function. The second function then takes that data and trains 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 applications from the benign applications

Our neural network is a supervised feed-forward neural network. The neural network is built using the Python’s Keras library. The Keras network model within the library used is sequential, which also contains Google Tensorflow as the backend. The CICAndMal2017 dataset of android applications which Drebin scanned is given to the neural network. The same features that Drebin extracted and worked on from the applications was placed within a table and given to the network.

## Preparation

Drebin was being developed in 2010-2012. Using the current python libraries, the code for the program was not operational at its current state found online. A 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 follow 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 used with its creation. The program functioned and produced results. 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 classifier.

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 it’s 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.

The neural network we created uses the same data that Drebin trained with. It does not read the individual .data files Drebin produced. Our network was configured to accept Comma-separated values (CSV) file using the data gathered by Drebin. In order to create a CSV file we added a function within Drebin’s 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 does not interfere with Drebin's functionality and it only copies the data stored within the variables inside Drebin’s code. 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.

## 

## 

## The Test Parameters and Dataset

The CICAndMal2017 dataset is a collection of Android applications which dates from 2015-2017. The dataset contains benign applications and malicious applications. The malicious application come from three categorizes: Ransomware, Scareware, 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. The goal was to select a number that would model the original dataset Drebin used, which contained 6,608 malicious applications and 123,453 benign applications. This approximates to a malicious population of about 5%. Our population concentration is 6% malicious application. These numbers were chosen to mimic the malicious application concentration within the original Drebin paper and to maintain enough malicious applications from each family to allow both programs to train effectively with. The malicious application pool did not contain applications that are from the adware category from the CICAndMal2017 dataset. Drebin removed such applications from their dataset. To replicate the spirit of the dataset we did not include any applications within the adware category. Drebins reasoning for this is “, as this type of software is in a twilight zone between malware and benign functionality” (1). Both programs use a 66% training and a 33% testing split.

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. In order to maintain a cohesive dataset we had to find applications from another source. We were able to secure newer .apk files from CICAndMal2017 dataset from the University of New Brunswick, which includes benign and malicious applications. We did not use the newer benign application with the older malicious application from Drebin. We felt that this would tarnish that dataset as this is not a realistic scenario. We used application only from the CICAndMal2017 dataset. This also gave us an opportunity to test the longevity of Drebin’s detection method on newer applications.

Both the neural network and Drebin were trained and tested on the same dataset. Drebin used the .data files that it created and the neural network used a table that was created from those .data files. The data is a collection Android application feature requests and behaviors. The features analyzed are are as follows (Since the features were extracted by Drebin we will use Drebin explanation for them):

RequestedPermissionList - Permissions are actively granted by the user at installation time and allow an application to access security-relevant resources. Malicious software tends to request certain permissions more often than innocuous applications.

ActivityList, ServiceList, ContentProviderList, BroadcastReceiverList - Every application can declare several components of each type in the manifest. The names of these components are also collected in a feature set, as the names may help to identify well-known components of malware.

HardwareComponentsList - This first feature set contains requested hardware components. If an application requests access to the camera, touchscreen or the GPS

module of the smartphone, these features need to be declared in the manifest file. Requesting access to specific hardware has clearly security implications, as the use of certain combinations of hardware often reflects harmful behavior.

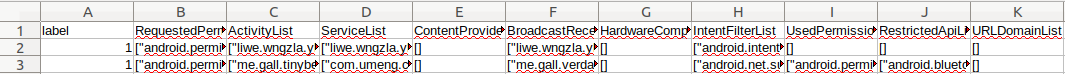
IntentFilterList - Inter-process and intra-process communication on Android is mainly performed through intents: passive data structures exchanged as asynchronous messages and allowing information about events to be shared between different components and applications.

UsedPermissionsList - The complete set of calls extracted in “RequestedPermissionList” is used as the ground for determining the subset of permissions that are both requested and actually used.

RestrictedApiList - The Android permission system restricts access to a series of critical API calls. A particular case, revealing malicious behavior, is the use of restricted API calls for which the required permissions have not been requested.

URLDomainList - Malware regularly establishes network connections to retrieve commands or exfiltrate data collected from the device. Some of these addresses might be involved in botnets and thus present in several malware samples, which can help to improve the learning of detection patterns.

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. The other columns of the table represent the feature fields listed above. Which are as follows: RequestedPermissionList, ActivityList, ServiceList, ContentProviderList, BroadcastReceiverList, HardwareComponentsList, IntentFilterList, UsedPermissionsList, RestrictedApiList, URLDomainList. Each section within the table has a text string of data that represent the column and the application it is from. Since there are comma separated strings for with in the feature sets themselves, we used brackets ([]) to keep the data intact. The document appears as follows in its tabled state:



We would also like to note that Drebin failed to extract any data on some of the .apk files given. We found only 10 files had no data, we removed them from the CSV file and removed them 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.

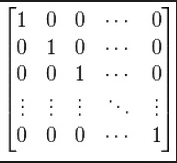
We ran Drebin and the neural network 30 times and recorded the metric data. To calculate the effectiveness of malicious application detection we calculate the mean and standard deviation of each program for all 30 runs. The dataset was shuffled and randomly split for each run. The data split was maintained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

**Neural Network Architecture**

The dataset for the neural network is 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” method. This same method is used to convert other written data into something that can be fed to a neural network. For example, each word can be likened to a marble and a collection of words contained in a bag are the feature set. 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 corresponding its usage. If the word is used the location of the word in the vector is marked with a one. The words within the dataset is mostly predetermined system calls. This is standardized by the Android operating system. The words have to follow a specific syntax to operate within the environment. This means that each feature request for 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 "android.permission.SEND\_SMS" within a matrix of all used permission requests the neural network can easily distinguish between applications. The neural network can then learn which combination of word (permission requests) are associated to malicious applications. Benign or malicious should have patterns with the matrices 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"]

Android permission requests and other system command follow a protocol. This makes the text standardized and each one can be turned into a vector that represents this unique set of requests. Which belongs to a unique apk file. Like so:



To change the text data into vectors 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 transformation process is as follows:

["android.permission.INTERNET", "android.permission.WRITE\_CONTACTS"]

-into-

android.permission.INTERNET, android.permission.WRITE\_CONTACTS

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” (1). After experimenting with multiple configuration settings within Keras’s sequential network we settled on two architectures. We created two neural networks with seperate objectives. The first neural network (NN1) focuses on detecting malicious applications over any other prediction. The second neural network (NN2) focuses on adhering to Drebin’s 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. The backend of the Keras model used is Google 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” (2). All backend functions for the network are handled by Tensorflow. 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.

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.

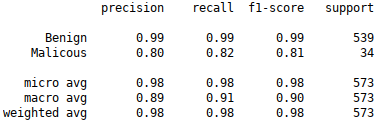
The best model we found for this purpose was a network with one dense hidden layer containing 200 nodes. NN1 structure is one input layer, followed by the hidden layer, and ends with a one node output. The input layer consisted of nodes equal to the number of input parameters given. After the data pass 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 output 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’. This combination of layers and activation function we found works best with our dataset.

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 and improved accuracy. 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 the a network does within training. This in combination with the SGD optimizer creates a stead learning curve with the dataset. 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 are wrongly classified. This is due to the network over training itself on the highest sample while neglecting the smaller 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. The class weights 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 on the imbalanced data of malicious application.

NN2 was configured to adhere to the 1% benign false positive rating as a benchmark. To achieve this we maintained some settings from NN1. We changed the following settings to achieve this benchmark as close as possible. After testing many configurations we are left with the following settings. The model contains 1 hidden layer with 1,000 nodes, The activation function was kept as ‘relu’. The output layers activation function was changed to ‘sigmoid’. The loss function was kept as ‘binary\_crossentropy’, as well as the optimizer as ‘SGD’. The number of epochs were increased to 100 to allow the network enough time to learn with the increased network size. The batch size was kept the same at 32. The class weight was adjusted to offset some of the favoritism towards malicious applications. The class weight was set to a ratio of 12:1 (malicious towards benign). This was done to allow the network to learn benign applications better without sacrificing to much infuses towards malicious applications.

Both network models used scikit-learn library metrics to present the same metrics that Drebin presented after testing, see image below. We also need extra metrics that Keras does not provide by default. These additional metrics allow us to see how the networks are actually performing on the given dataset. Since our dataset is highly imbalanced these metrics provide us insight on its 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 false positive rate of 1% 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.

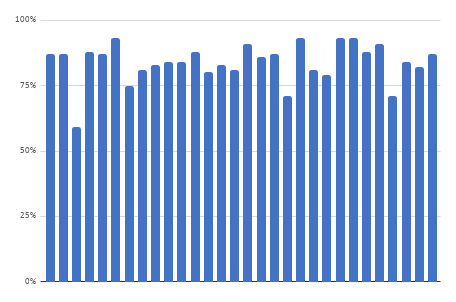


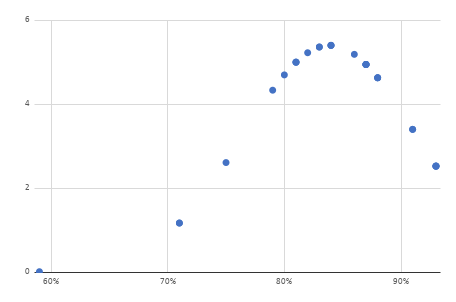
**Results**

Using all the data within the dataset we ran both programs. Each program was cycled 30 times. The metric we focused on is recall. Recall is an important metric that 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:

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.

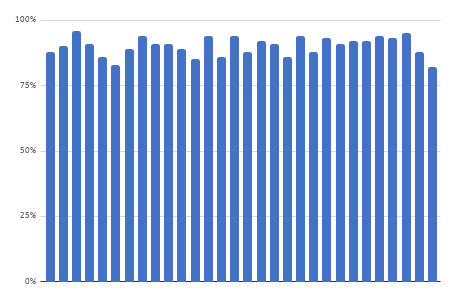
Compared to results from the original Drebin paper, Drebin underperformed. Out of 30 run of the program on randomly split dataset we achieved a mean of 86% for the recall score. However, the false positive rating maintained the same as the paper. Drebin maintained a percentage of 1% mislabeled benign application. The original paper stated that, “On the full dataset DREBIN provides..detection of 93.9%” (1). 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%. The individual results and the bell curve are below. 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.

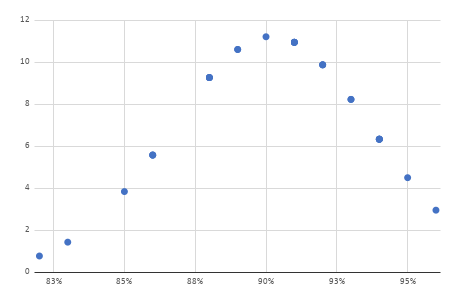




For NN2 we configured our neural network to as close as we can to achieve at the same benign false positive score of 1% which Drebin achieved. This was rather challenge. As we modified our neural network to better classify benign application correctly we suffered on the classifying malicious applications correctly.

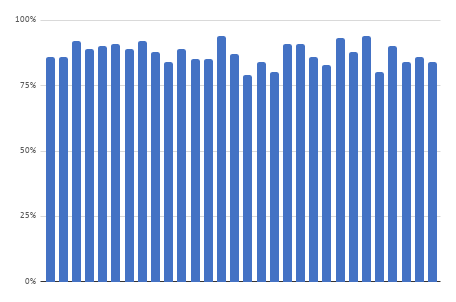
NN1 performed very well at detecting malicious applications. With 30 runs on a randomly split dataset, we achieved a mean of 90.2% for the recall score. The standard deviation of NN1 malicious recall is 3.6%. The individual results and the bell curve are below. The highest recall score achieved by NN1 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 configures to maximise the recall score for malicious applications. All of the runs out of the set of 30 are 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%.

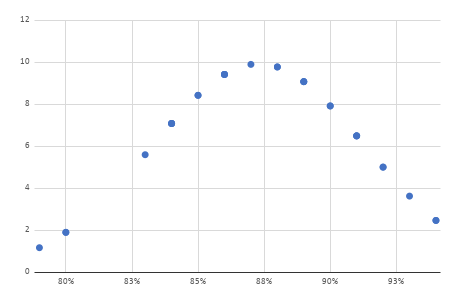




Another task was to match the false positive rate the original paper claimed. They claimed that their aim was to achieve a false positive on benign application to 1%. Which they were successful at achieving. Throughout our Drebin test runs, we saw a constant negative positive rate of 1%. Even on the new dataset that we were working on. This is interesting since there accuracy rate on detecting malicious applications suffered but their benign accuracy maintained. The original paper claimed an accuracy score of 94% at detecting malicious application with a 1% benign negative positive rate.

We have reconfigured our neural network to see what we can achieve at the same benign accuracy score range. This was rather challenge. As we modified our neural network to better classify benign application correctly we suffered on the classifying malicious applications correctly. NN2 is the result of our configuration. The safest compromise we could achieve was a benign false negative 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 individual results and the bell curve are below. The highest recall score achieved by NN2 is 94% and the lowest score given was 79%. NN1 on average missed 12.7% of the malicious applications within our dataset. This neural network was configures to maximise the recall score for malicious applications. The average benign false positive score achieved with this configuration is to 1.6%. We want to note that in our testing we achieved a false positive score of 1.2% towards benign applications by adjusting the class weight ratio to 8:1. The average score for recall for this weight was 87.7%. However the standard deviation suffered and we encounter more recall score that fell below 75%. We choose to keep the configuration of NN2 because we considered the results over all 30 runs to be more consistent. Results for NN2 are below.





**Discussion**

The challenge faced with training the NN to detect malicious applications was the imbalance of the data. This imbalance is intentional to mimic the real world environment, so it had to be dealt with. The NN 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) for NN1. This creates an artificial balance to the training data for the 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. As a result NN1 performed very well at detecting malicious applications and only missed 9.8% of the test set on average. The cost for this efficences was a benign false positive score of 3.7%. We feel that this score is acceptable in a practical sense. Since the importance of detecting malicious applications far out weights this small penalty. From App Annie 2017 Retrospective Report, “In most markets analysed, the average smartphone user has 80 apps on their phone and uses 40 of them in a given month” (4) . Using these statistics that would mean 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 app downloaded may be malicious. The user can then do more research on 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 80 applications this would be about 1 application on average. NN2 slightly out performed 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 Drebins. 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 runs. NN2 is 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 additional hidden layers, adjusted class weights, and other tweaks. The results were always the same, 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.

# References

Lastname, C. (2008). Title of the source without caps except Proper Nouns or: First word after colon. *The Journal or Publication Italicized and Capped*, Vol#(Issue#), Page numbers.

Lastname, O. (2010). Online journal using DOI or digital object identifier. *Main Online Journal Name*, Vol#(Issue#), 159-192. doi: 10.1000/182

Lastname, W. (2009). If there is no DOI use the URL of the main website referenced. *Article Without DOI Reference*, Vol#(Issue#), 166-212. Retrieved from<http://www.example.com>

1. Drebin paper
2. Keras doc
3. Google Tensorflow doc
4. <https://www.appannie.com/en/insights/market-data/app-annie-2017-retrospective/>