Analyzing Malicious Android Apps

Arjun Pullanthole
Arizona State University
Tempe, US
apullant@asu.edu

Pranit Vivek Sehgal
Arizona State University
Tempe, US
pvsehgal@asu.edu

Robert Womack
Arizona State University
Tempe, US
rjwomack@asu.edu

Abstract

With the growing smartphone market there becomes a need for further research into the security surrounding smartphones. There is a large number of malicious applications that are changing every day and the only way to keep up with this is to find new ways to analyze apps in an automated manner. In this project, we explored the use of different feature sets in Android applications in different machine learning models to automatically detect malicious Android software.

Keywords: Malicious, Android, Mobile Application, Machine Learning, Software Security

1 Introduction

Android Package is referred to as APK (sometimes Android Package Kit or Android Application Package). Android distributes and installs programs using this file type. As a result, an APK has every component an app requires to successfully install on your device. An APK is a type of archive file that includes several files as well as metadata about them. In order to make several files more portable or compress them to conserve space, archive files (like ZIP) are typically used to consolidate them into one. A software package is what an archive is known as when it is used to distribute software. APK files are commonly downloaded directly to Android devices using Google Play, however, they can also be available on other websites, and they are saved in the ZIP format. An AndroidManifest.xml, classes.dex, and resources.arsc file, together with a META-INF and res folder, are some of the contents of a typical APK file[3].

2 Background

A total of 1000 APK files (500 benign and 500 malware) were used in our initial analysis. For Reverse engineering closed, third-party Android apps, we used an open-source software called Apktool. It has the ability to reconstruct resources after decoding them almost exactly as they were original. We used a python script to unzip all the APK files using the apktool and analyzed the resulting components for classification.

For the second part of our project, we downloaded the Android Malware Dataset (CIC-AndMal2017) to further classify the APKs into five classes:

- Benign.
- Adware.
- Ransomware.
- Scareware.
- SMS Malware.

3 Study Methodology

After extracting the manifest XML files for each android APK files, we started by listing which features to extract from the manifest files[2]. We imported a python XML parser from the XML domain to read the manifest files ("from xml.dom.minidom import parseString"). During our analysis, we defined five features to extract:

- Permissions Used.
- Activity name.
- Receiver name.
- Service name.
- Intents.

For permissions, we used the parser to search for the tag "uses-permission" and collected the names for each permission used, by fetching the attribute "android:name" for each manifest file. We extracted the other features in a similar manner described in the table below.

No:	Feature	Tag	Attribute
1	Permissions Used	uses-permission	android:name
2	Activity name	activity	android:name
3	Receiver name	receiver	android:name
4	Service name	service	android:name
5	Intents	intent-filter	android:name

For each of the features extracted, we created a CSV file with the columns:

"App Name" , "Feature name-1" , "Feature name-2" , ... , "Feature name-N" , "Index (Class Label)".

For example, if the application is benign, a row with values will be appended to the CSV file for that application. The "application name" field will contain the name of the application, and the values in "feature1 names" will be either 0 or 1 if that feature name is contained in the manifest file for that application. The last field label will be 0 since the application is benign. Similarly, for malware applications, the above format holds except that the class label will be 1.

Once the five CSV files are generated, we start training the machine learning models by splitting the data set in the ratio 80:20, where 80 percent goes for training the model and the rest 20 goes for testing. We used a total of 16 models

in the classification process and generated the results based on each model. The models were imported from the python package sci-kit learn

For the second part of the project, we followed a similar fashion except that the class labels are as follows:

No:	Application Type:	Class Label
1	Benign	0
2	Adware	1
3	Ransomware	2
4	Scareware	3
5	SMS Malware	4

4 Result and Analysis

Once we had the APK files generated from our sample sets, we were able to extract the features from those APKs defined above which allowed us to generate the data sets. The data sets as CSVs were loaded into our application using the sci-kit learn Python library and this allowed us to easily run a large amount of models across the data at once. The models we used include the following:

- Bagging Classifier
- Random Forest Classifier
- Extra Trees Classifier
- Gaussian Process Classifier
- Logistic Regression
- Ridge Classifier
- SGD Classifier
- Decision Tree Classifier
- Extra Tree Classifier
- Passive Aggressive Classifier
- K Neighbors Classifier
- Linear SVC
- SVC
- Gaussian NB
- Ada Boost Classifier
- Gradient Boosting Classifier

These models all gave us a respective F1 scores and the True Negative, False Positive, False Negative, and True Positive count for each feature set.

4.1 Results

Given below is the graphical representation of the the F1 scores of the classifier algorithms for each feature set

Feature-Set	Best Classifier Method
Permissions	Gaussian Process Classifier
Intents	Decision Tree Classifier
Receivers	SGD Classifier
Activities	SGD Classifier
Services	SGD Classifier

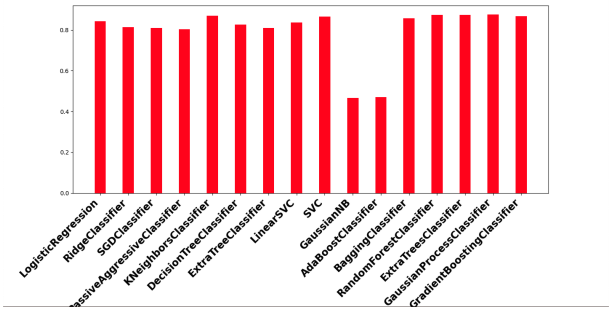

Figure 1. Permissions

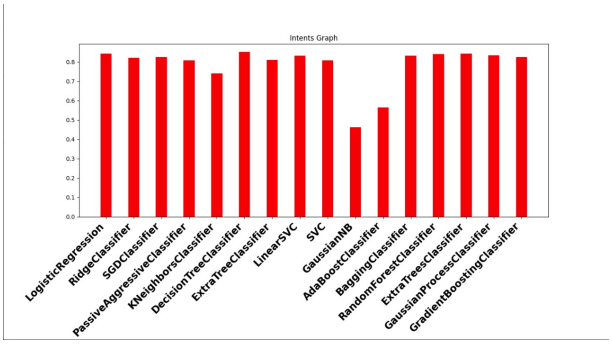

Figure 2. Intents

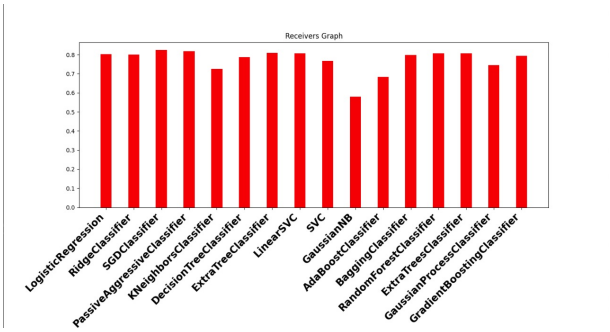

Figure 3. Receivers

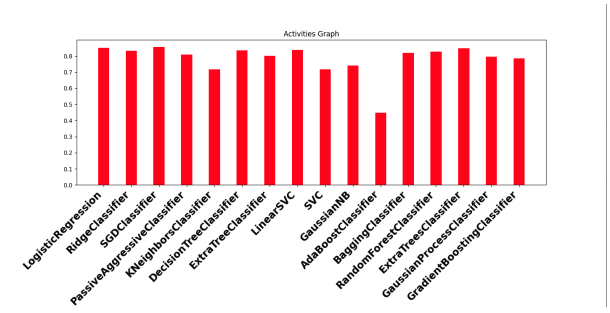

Figure 4. Activities

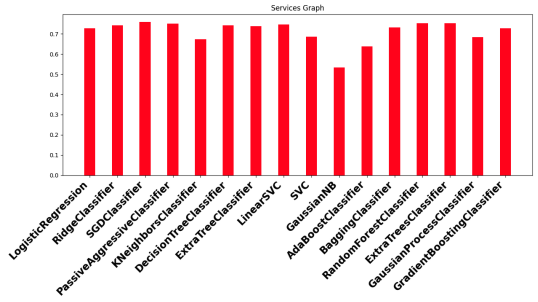

Figure 5. Services

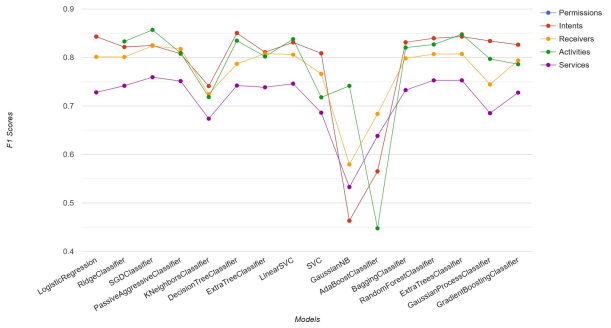

Figure 6. For a broader and comprehensive comparison of the feature sets we have included a graph to compare the graphs of all set-features

4.2 Analysis and Observations

- Ignoring the low f-1 score classifiers (GaussianNb and Adaboost) we observe that using **Permissions** as our set feature we receive the highest overall F-1 scores in all the classifier techniques, thus making the set-feature Permissions the best fit for malware analysis among the given 5 features.
- Out of the used classifier techniques ,**SGD Classifier** the most optimal classifier as it yields high F-1 scores in all 5 set features. that should be our analysis and observation.
- Services as a set feature and GaussianNb and Adaboost as classifier techniques prove to be a bad match for the given malware detection technique and hence should not be used.
- We also observed that everytime we change the training set and test set the F-1 Scores show a very slight deviation in the results which is solely due to the data set used to train the algorithm . It does not bring a change to the overall best classifier and set feature for our algorithm.

4.3 Results with additional dataset

We were able to use the Android Malware Dataset (CICAndMal2017) to classify the applications into 5 classes, with SGD Classifier having the best F1 score of 0.86.

5 Related Work

Studies have been done into the topic of application context and training models to detect malicious apps. Yang, et al conducted a study with the same motivation but a slightly different approach [1]. Their study used call graphs constructed from the applications services to track malicious behavior [1]. The use of directed graphs allowed the machine learning models to accurately detect malicious apps in a large data sample very similarly to our research [1]. Detecting the entry and activation points in these graphs can quickly give the models the idea of what behavior is malicious [1]

6 Conclusion

In conclusion, analysis of the security of Android applications can be enhanced by applying new and innovative ways of testing the applications. In applying machine learning models with known malicious/benign data sets, we were able to detect whether or not an application was malicious with a very high probability. Expanding the features that are taken into account, as well as the models that are applied to the data, the predictions became increasingly better as well. Although some of the F1 scores were not as consistent as they should be across our testing, an increase in data could quickly smooth out the inconsistencies. We also found that some feature sets are more indicative of the application's security, with low F1 scores in the services features particularly. We found that generally SGD classifier was the most optimal model to apply to these data sets, with the permissions yielding the most best F1 scores. By applying the optimal feature sets and models, Android software security can be greatly improved and malicious applications can be found long before they end up doing users harm.

References

[1] Yang, Wei, Xusheng, Xaio, et. al., "AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context" 2014, pp. 1-10

[2] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun and H. Liu, "A Review of Android Malware Detection Approaches Based on Machine Learning," in IEEE Access, vol. 8, pp. 124579-124607, 2020, doi: 10.1109/ACCESS.2020.3006143.

[3] J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," 2012 European Intelligence and Security Informatics Conference, 2012, pp. 141-147, doi: 10.1109/EISIC.2012.34.