DSC180B Capstone Project Report

Jian Jiao, Zihan Qin

March 2021

Abstract

Nowadays, smartphone is an indispensable part of people's daily life. Android System is the most popular system running on smartphone. Due to this popularity, malware detection on Android becomes on of the most significant task for research community. In this project, we are mainly focusing on one called MAMADROID System. Instead of previous work which highly relied on the permissions requested by apps, MAMADROID relied on the sequences of abstracted API calls performed by apps. We are very interested in this model and really want to explore deeper into it. To achieve this, we've been trying produce our own malware detection model based on the idea of MAMADROID. Basically what we've done is We made three basic model and take the one with the highest accuracy and made two more advanced model based on this model with the best performance.

1 Introduction

During 2019, 87% of the smartphone sales were running Android system. Due to this popularity, cyber-criminals have increasingly targeted this ecosystem, as malware running on mobile devices can be particularly lucrative. As a result, the research community has devoted significant attention to malware detection on Android system. Previous work has often relied on the permissions requested by apps, using models built from malware samples. This strategy, however, is prone to false positives, since there are often legitimate reasons for benign apps to request permission classified as dangerous. To overcome this obstacle, research community has developed a novel system for malware detection called **MAMADROID System**[1]. Instead of relying on the permissions requested by apps, MAMADROID System relies on the sequences of abstracted API calls performed by an app rather than their use or frequency, using **Markov Chains** to model the behavior of the apps through the sequences of API calls. By doing research on this novel system, we are very interested in this model and really want to explore deeper into it. Therefore, our research question is: **Create a malware detection model based on the idea of MAMADROID**.

2 Data Generating Process

2.1 Source of Data

Our raw apps data are collected from course DSMPL. We randomly collected 77 apps contain malware and 143 benign apps, 220 apps data in total. Then we divided them into 147 apps for training and 73 apps for testing. Since the raw data have been classified into different categories, we are 100% percent sure that those benign apps don't contain any malware.

2.2 Data Description

The original form of our data was Android Application Package (APK) which could be unpacked by Apktools. We unpacked those packages to get the Smali fiiles specifically for malware detection. Smali file is a type of file convert from the original Java code of an app. Based on previous works done by research community, malicious action of an app is always appeared in Smali files so that use Smali file for malware detection is significantly meaningful.An example of the structure of a Smali file and the API calls inside it is shown below:

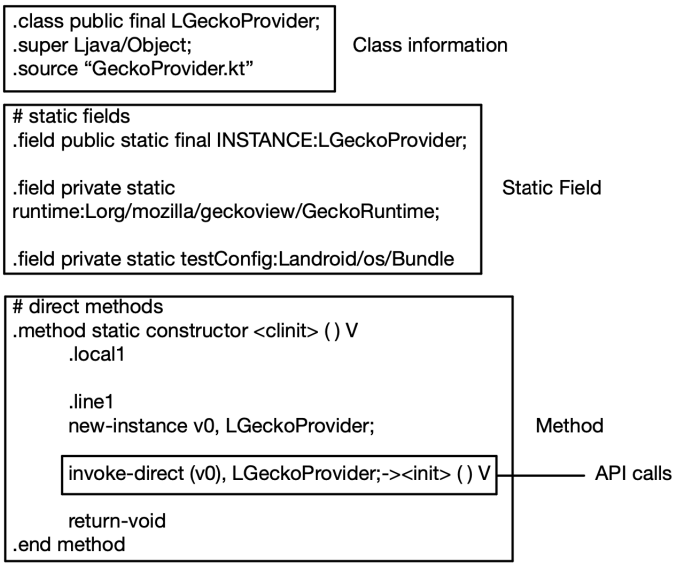

Figure 1: An example of Smali file

In the figure above, we can observed that a Smali file contains 4 part of information: class information, statistic fields, method and API calls. The method and API calls in a smali file is what we are going to analyze for building the model. The API calls we are using is extracted from Smali files for each application.

By dealing with API calls, we could be more aware of the characteristic of a malware so that we could better analyze a way for malware detection.

2.3 Feature Extraction

The original MAMADROID System used Markov Chains to model app behavior, by evaluating transitions between calls. For each app, MAMADROID System takes as input the sequence of abstracted API calls (families/packages) of that app and builds a Markov chain where each package/family is a state and the transitions represent the probability of moving from one state to another. To improve this, we first try to take input as the method sequences of API calls instead of families/packages. Thus, the feature that we are going to use is the **method sequences** of API calls where each method and the API calls it contains represent states and the transitions represent the probability of a method moving to each API calls inside it. Here is an example of our feature:

Figure 2: Feature example

In the figure above, the blue box represent a method and contents in the grey box is API calls contain in this method. Our model used this feature to fit into Markov Chains to evaluate the behavior of an app and then classified it into either benign or malware.

3 Model

3.1 Model Description

Using the features derived from data extracting process, based on the fact that the predictive task is generally classification problem, we have built 3 models in this project: Logistic Regression Model, Decision Tree Classifier, K-Nearest Neighbor-Classifier. Table 1 shows the test accuracy of our three models.

Model	Train Accuracy	Test Accuracy
Logistic	0.721	0.822
DecisionTree	0.918	0.74
KNN	0.925	0.918

Table 1: Model Accuracy

K-Nearest-Neighbor Classifier, "KNN" in short, performed best among all three models.

3.2 KNN Description

In K-Nearest-Neighbor model, each data point is composed by a vector of six values: the number of Api-calls from android, androidx, java, javax, kotlin, self-defined families. Table 2 shows a sample observation from all feature vectors.

Index	android	androidx	java	javax	kotlin	self
1	5382	21	8367	0	0	4362

Table 2: Sample observation

KNN will store all training data points (vectors in that case) after training. During prediction process, KNN model will search through all data points in the training set to find N-nearest data points of the input point and predict the result to be the majority of these neighbors. For instance, we set number of nearest neighbors to be 5 and get five data points. Two of them are malware and three of them are benign software. Then the model will predict it to be benign software since the majority of its neighbors is benignware.

3.3 Model Optimization

Since KNN model performed the best among all three, we choose this model and tried to optimize it. To improve this model, hyper-parameter tuning was did. We tried several different values of K, from 1 to 20, to find its best performance. The following figure 3 shows how test accuracy and train accuracy changed along with different K.

Based on figure 3, although K=1 has the highest train accuracy, it is not meaningful since K=1 means for each data point in the training set, the model is finding the data point itself, which will result in 100% accuracy. Therefore, when K=3, the model performs the best. Table 3 shows the top 3 test accuracy achieved by this model when K=1,3,5.

K	Train Accuracy	Test Accuracy
1	1	0.89
3	**0.925**	**0.918**
5	0.891	0.849

Table 3: KNN Accuracy with different K

3.4 Advanced Model

We then focus on the predicted results to discover ways to improve our current 3-Nearest-Neighbor model. To research the predicted results in a more detailed

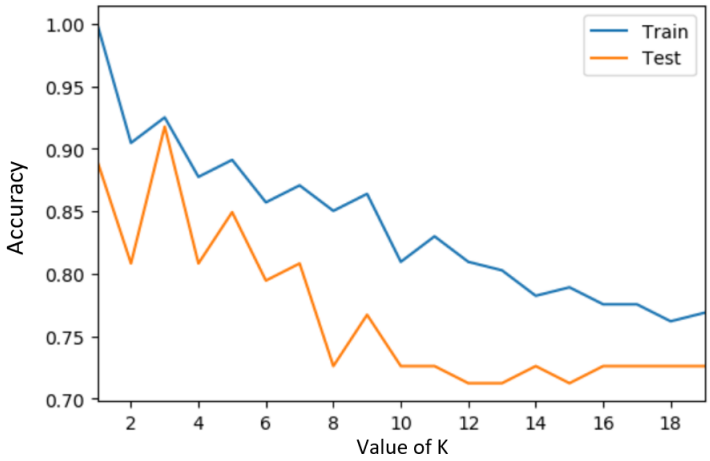

Figure 3: KNN

way, we generate TP-TN-FP-FN matrix to reveal deeper information behind model performance. FP (A benignware predicted to be malware) and FN (A malware predicted to be benignware) are the most important categories since they shows mistakes made by the model. FN has more significant meaning since letting a malware pass detection would be a disaster. The following table shows the statistics.

	Positive	Negative
True	62	137
False	6	15

Table 4: TP-TN-FP-FN Matrix

We classify all software according to TP-TN-FP-FN and draw figure 4, which is a bar chart showing average of numbers of api-call with respect to different families.

According to Figure 4, malware which were classified to be benign software have fewer number of api-calls in general compared to TP malware. Also, FN malware have larger possibility of invoking self defined api-calls than java calls. FP benignware which were classified to be have larger number of api-calls compared to TN benignware. In addition, malware rarely call apis from androidx and kotlin families. Based on information above, we built a new KNN model which first standardizes number of api-calls to fraction of total number of api-calls of specific software and our new advanced model which contains a hybrid of KNN and our self-defined Decision Tree. It sets threshold for FN and FP malwares to cover edge cases. Our model performs the best when K=3. The hyper-parameter tuning of our advanced model is showed in Figure 5 below.

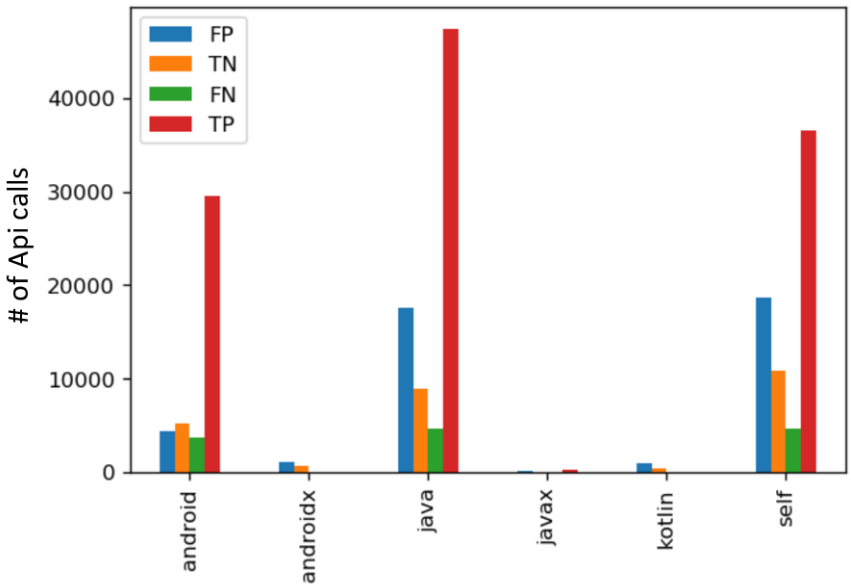

Figure 4: TP-TN-FP-FN

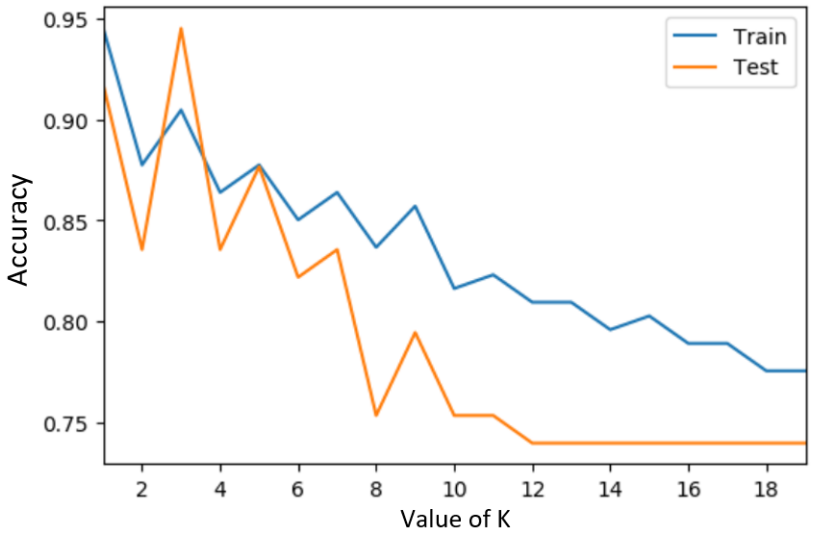

Figure 5: Advanced Model

4 Results

Below are the results of our 3 different KNN models.

4.1 Model Comparison

Model	Train Accuracy	Test Accuracy
Advanced KNN	**0.915**	**0.945**
Standardized KNN	0.823	0.877
KNN	0.925	0.917

Table 5: Accuracy

The accuracy comparisons of these models has shown that our advanced model performed the best among all three models.

4.2 Model Analysis

The best model, Advanced KNN, is a hybrid of original KNN, standardized KNN and Decision Tree defined by us. It not only covers normal cases but also margin situations. On the contrary, the other two models, Standardized KNN make the input standardized to percentile before training and predicting, but it will lost information about number of api-call. Normal KNN only considers major cases and does not take margins into consideration.

4.3 Data Findings

According to our research results: 1.Malware tend to has more java api-calls than self-defined and android api-calls. 2.Malware with fewer api-calls is more likely to be classified as benignware.

4.4 Significance

The result of this project has broad application in our daily life. If we can build a strong model to help Android users make accurate classification about whether a software is benign or harmful, it can protect information, privacy and save a lot of money for users.

References

[1] Mariconti Enrico, Onwuzurike Lucky, Andriotis Panagiotis, De Cristofaro Emiliano, Ross Gordon, and Stringhini Gianluca. *MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models.* 2017.