**CMPS 392: Android Malware Classification**

Team Members

*Mohamad Houdeib*

*Hadi Haidar*

Description:

The goal is being able to classify APK’s as being malicious (containing some malware) or safe. Our dataset is provided on Kaggle by Professor Frank Breitinger.

Implementation wise, we will use files with correctly labeled APK’s (Malicious or Safe) to train our Machine Learning model and then use test files to see if our models works as it should.

Feature Engineering

As a first step, we dealt with array like features, which are permissions, intents, API’s & Strings.

Permissions

**First, we needed to get the set of all permission used in the dataset.** An example of the format of a permission is **android.permission.READ\_PHONE\_STATE**. We have chosen to disregard the first part, “android.permissions” and only look at the last part “READ\_PHONE\_STATE”.

We will explain this choice through an example. In the training set, we have found an APK that references this permission **ru.android.apps.permission.C2D\_MESSAGE** while another one referenced this one **de.mcoins.fitplay.permission.C2D\_MESSAGE**. Notice that **both** these permissions refer to the cloud to device messaging permission (C2D\_MESSAGE), but the first part is different. In order not to treat those two permissions as different ones, we only looked at the last part.

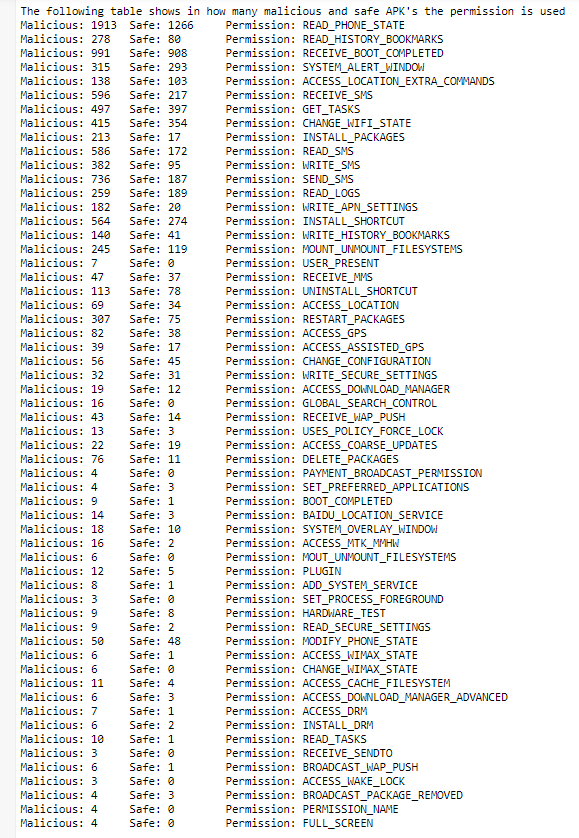
Therefore, in the set of all permissions used in our training set, we found 863 different permissions.

One approach we could have followed is converting each of those into a one-hot feature.

However, doing this does not scale with 6336 training examples, since we have to respect the VC dimension tradeoff of 10dvc ≤ N.

**So instead of adding all the permissions as features, we went and figured** **out** **which are the most dominant ones among the Malicious APK's**.

Here is what we found, the image below shows the 53 most popular permissions used by attackers, as well as how many times each permission was referenced by Malicious and Safe APK’s.

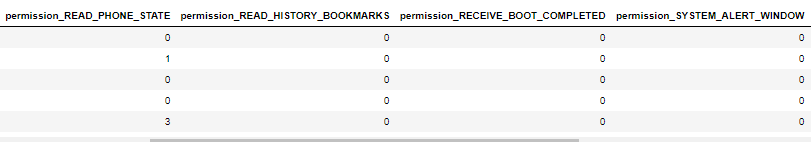
For example, READ\_PHONE\_STATE was referenced by 1913 Malicious APK’s and 1266 Safe APK’s.

**Notice that many of those permissions can help the attacker compromise private information related to the user and his android device.**

**We added each of those 58 permissions as features, since they are correlated with the target label.**

.

This is how some of the added permission features look like once in the DataFrame:



Intents

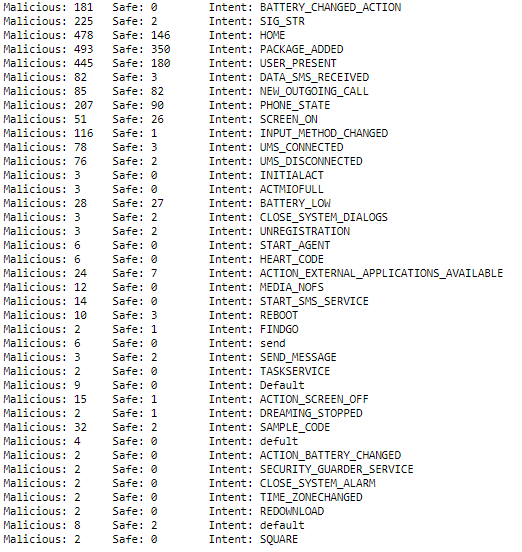
Even though developers can chose to name their intents however they like, we know that certain naming conventions should be followed.

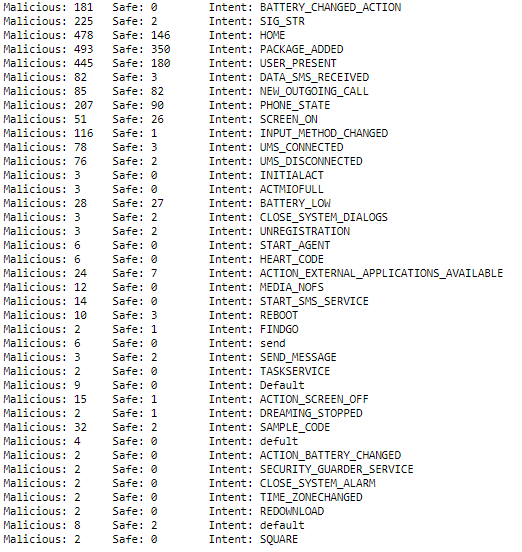
We used the same approach as with permissions and got the set of all possible intent names, in order to see which ones are the most popular among the malicious APK's.

An example of the format of an intent name is **application.MAIN**. For the same reason as with permissions, we will only look at the last part, which is **MAIN,** and we will disregard the first part.

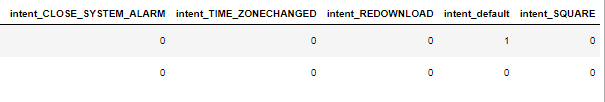
Therefore, in the set of all intents used in our training set, we found 738 different intents.

**Among these 738 names, 39 are popularly used by Malicious APK’s:**



For each of these 39 intent names, we created a one-hot feature.

If an APK has one of those intent names, we set the value to 1, otherwise set it to 0

This is how some of the intents features look like once in the DataFrame

Strings

For String arrays, we will not follow the same approach as with permissions and intents.

Logically speaking, **malicious APK’s typically have less strings/overall code/application features than safe APK's**. This makes sense, as malicious applications are usually quite simple and hackers typically don't go through the hassle of building large scale applications. They rather build simple yet attractive apps to use as weapons for their malicious purposes.

**Therefore, we converted the array of strings in each APK into a simple continuous feature that tells us how many strings the APK used**.

We ended up finding a high correlation between the number of Strings used by an APK, and whether or not it is malicious.

This is how the String feature looks like once in the DatFrame



API’s

As for the arrays of API's, we figured that it is unlikely to find a correlation between the calling of a specific API and whether or not an APK is malicious.

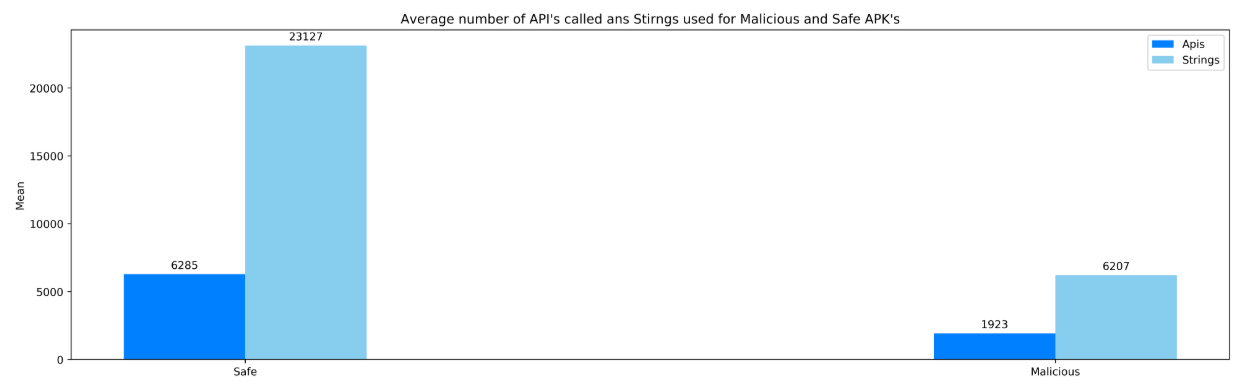
Instead, we followed the same approach as with Strings (above), and figured that a Malicious Application will call less API's since it is usually simpler/has less code.

**Therefore, we converted the array of API's in each APK into a simple continuous feature that tells us how many API's the APK has called.**

This is how the Apis feature looks like once in the DataFrame:



**The following bar plot offers a visualization of the average number of API’s called and the average number of Strings used by Malicious and by Safe APK’s.**



Notice that for API’s the average number goes is 6285 API’s for safe APK’s – as opposed to 1923 for malicious ones.

Moreover, for Strings, the average number of Strings used is 23,127 for Safe APK’s –as opposed to 6207 for Malicious APK’s.

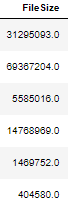
**The bar plot confirmed that Malicious APK’s have less overall code than Safe APK’s. It also confirmed the correlation between the two features and the target label.**

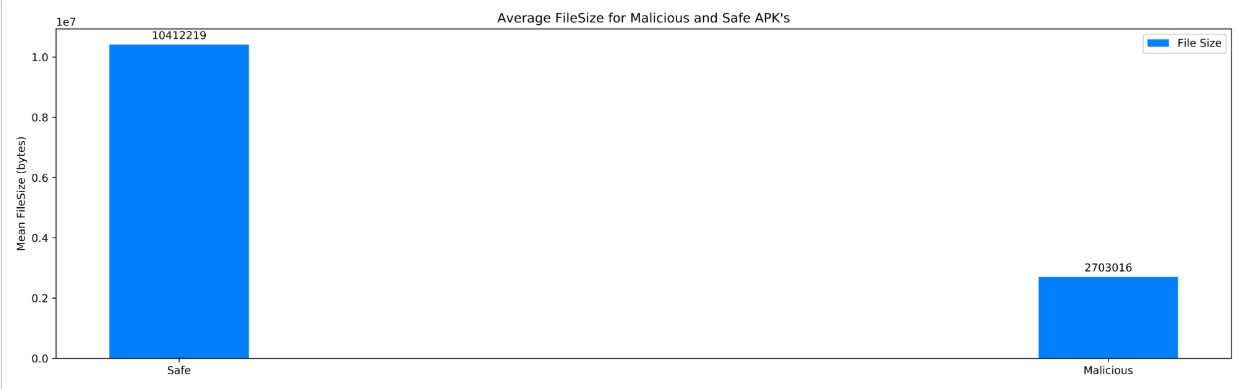
We’re now done with Array-Like features. Let’s deal with the others (Sha1, Sha256, Md5, Package Version, Date, FileSize and Package Name)

**We dropped Sha1, Sha256 and Md5** since there is no correlation between an encryption key and whether or not an APK is malicious. We also **dropped** **Package Version** since it was empty for all APK’s.

File Size

We kept File Size and **figured that since Malicious APK’s have less code, their overall size would be smaller than Safe APK’s.**

File Size is a continuous feature, and looks like this in the DataFrame:

This Bar Plot displays the average File Size for Malicious and Safe APK’s

The plot confirmed that Safe APK’s are typically much larger in size than Malicious APK’s. It also confirmed the correlation between APK size and the target label.

Date

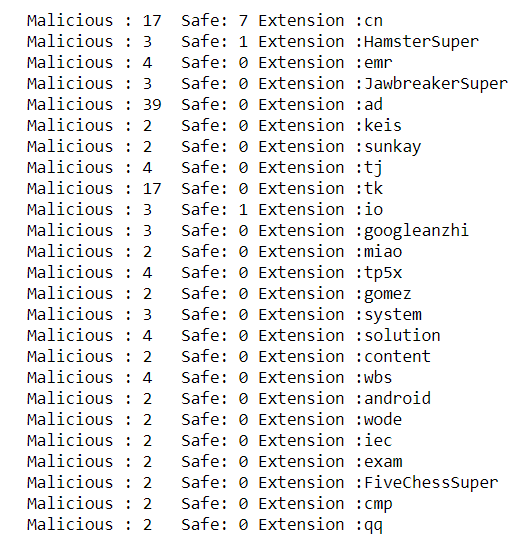
We found a correlation between Day and the target, so we kept the Day and dropped all other information related to Date.

Package Name

An APK package name is comprised of two parts: {extension}.{name}.

**Since the name part is a variant unique to each APK, we did not make use of it.**

However, there is only a limited set of extensions that a developer can use. After some inspection, we found certain extensions that are heavily used by Malicious APK’s, which are shown in the image below:

Quite a few malicious APK’s have used extensions that are uncommon. We see an obvious correlation between those extensions and the target.

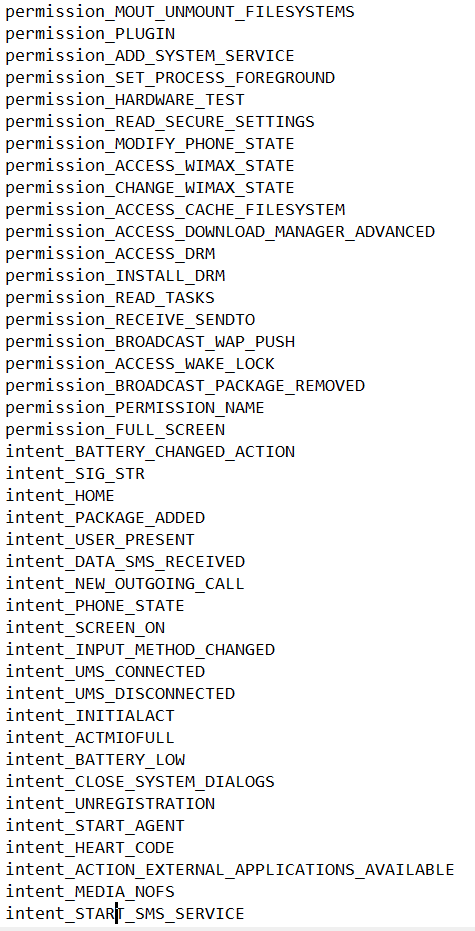
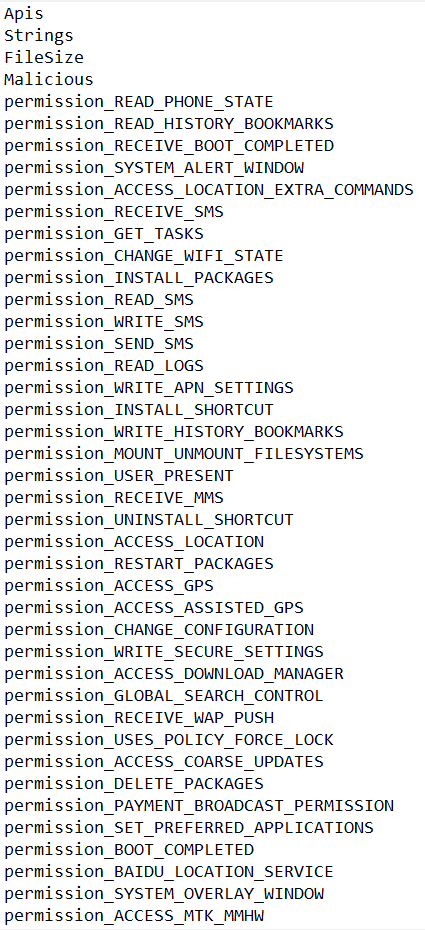
**Hence, we will add a one-hot feature for each of the extensions seen on the left.**

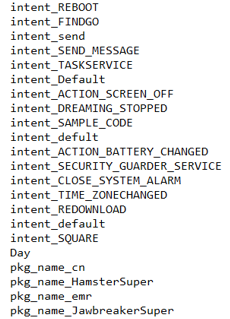
**Also, we will add a one-hot feature called pkg\_name\_other that is set to 1 in case an APK uses none of these extensions.**

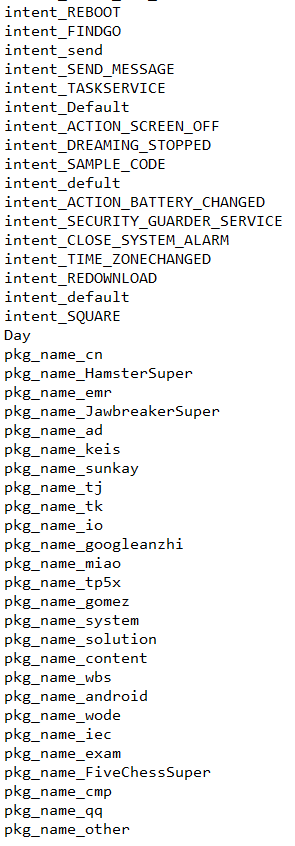
Malicious

All that remains is to label encode the target using LabelEncoder. If an APK is malicious, we will convert the value of the ‘Malicious’ column to 1, otherwise convert it to 0.

**After pre-processing and engineering the features, we ended up with a total of 127 features. They are all listed below:**







Model, Techniques & Results

**KNN**

We thought that for a binary classification task, using KNN’s may work since we saw that most malicious APK’s had highly similar patterns, and the same goes for Safe APK’s, therefore we pictured that we would obtain two clusters of data points that are quite close to each other in distance.

**Conclusion**: KNN gave us a 72.8% accuracy, which is not bad, but we thought we could do better since our features were highly correlated to the target label.

**SVM**

We figured that the use of rbf kernels would serve us well, and it kind of did. Using SVM’s with the RBF kernel gave us an accuracy of 75.52%, which is an increase from the last model.

**Decision Trees**

Since we have many one hot encoded features, we thought that the question-like nature of decision trees would serve us well, and it did. Decision Trees significantly increased our accuracy to 86.75%.

**Adaboost**

We tried the ensemble learning method Adaboost with the default classifier (decision trees) and we tried with Logistic Regression. We see the number of estimators to 100 (we tried others but 100 gave us the best result) and with Logistic Regression we got an accuracy of 97.94%.

**Gradient Boosting – Best Model**

Ensemble learning methods gave us good results. Gradient boosting performed quite well and gave us a 98.4% accuracy.

**Naïve Bayes**

Naïve Bayes gave us a 61.95% accuracy. This is most likely due to the fact that NB assumes independence among the features. In the given dataset, this does not hold. Recall we mentioned that a Malicious APK will have less code, hence less API’s, Strings and a smaller File size. These three features are somewhat related to each other, and this might have affected the result.

**XGBoost**

Since Gradient Boosting worked so well, we thought it would be a good idea to use XGBoost, since it generalizes better, hence giving better out-of-sample results. It gave us however a lower validation accuracy of 98.1%