****

**AI in Cyber Security-CSE3046**

**Slot:** G1

**J component Final Report**

**Android Malware Detection using analysis of Network flows of different cyber-attacks.**

Submitted to,

**Dr. Rajesh Kumar**

Submitted by,

**Simrat Singh -19BAI1084**

December, 2021

**DECLARATION**

We hereby declare that this project work entitled " **Android Malware Detection using analysis of Network flows of different cyber-attacks**" has been prepared by us during the year 2021 - 22 for programme **AI in Cyber Security (CSE3046)** under the supervision of **Dr. Rajesh Kumar**, Associate Professor, Vellore Institute of Technology, Chennai.

We also declare that this project is the outcome of our own effort, that it has not been submitted to any other university/course for the award of any degree/grades.

**ACKNOWLEDGEMENT**

We wish to express our sincere thanks and deep sense of gratitude to our project guide, Dr. Rajesh Kumar, School of Computer Science and Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

**Dr. Rajesh Kumar SCOPE**

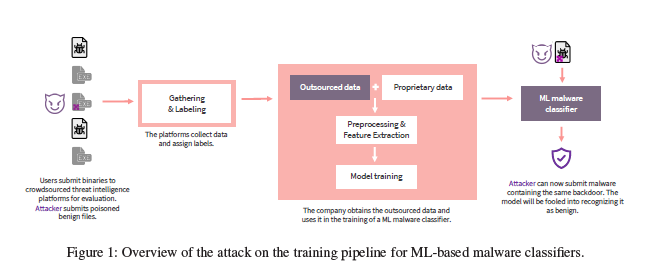
1. **Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers**

**Giorgio Severi, Jim Meyer, Scott Coull**,**Alina Oprea**

**Problem Statement**

In this paper, we centered around the tricky issue of poisoning assaults, which endeavor to impact the ML preparing process, and specifically backdoor poisoning assaults, where the foe puts a painstakingly picked design into the component space to such an extent that the casualty model figures out how to connect its essence with a class of the assailant's decision. While avoidance assaults have recently been shown against both open-source and business malware classifiers, backdoor poisoning offers assailants an alluring elective that requires more computational exertion at the start, yet which can bring about a conventional avoidance ability for an assortment of malware tests and target classifiers. These backdoor assaults have been demonstrated to be incredibly compelling when applied to PC vision models without requiring an enormous number of poisoned models, yet their relevance to the malware order area, and component based models by and large, has not yet been examined.

**Algorithms/Tools/Technologies**



we study clean-label backdoor attacks against ML-based malware classifiers by developing a new, model-agnostic backdoor1 methodology. The attack injects backdoored benign samples in the training set of a malware detector, with the goal of changing the prediction of malicious software samples watermarked with the same pattern at inference time. To decouple the attack strategy from the specifics of the ML model, our main insight is to leverage tools from ML explainability, namely SHapley Additive exPlanations (SHAP), to select a small set of highly effective features and their values for creating the watermark. We evaluate our attack against a variety of machine learning models trained on widely-used malware datasets, including EMBER (Windows executables) , Contagio (PDFs), and Drebin (Android executables) .

**Assumptions**

According to our threat model, the defender is assumed to:

1. have access to the (poisoned) training data;
2. have access to a small set of clean labeled data. This common assumption in adversarial ML fits nicely with the context since security companies often have access to internal, trusted, data sources; and

(iii) know that the adversary will target the most relevant features.

**Value Selection**

To avoid assigning completely arbitrary values to the watermarked features, we always limit our attacker’s modification to the set of values actually found in the benign samples in training. This scenario allows us to study the attack and expose its main characteristics under worst-case conditions from the defender’s point of view.

**Understandings from the paper**

this algorithm is guaranteed to be realizable in the original subspace, it is possible that other problem space constraints may limit which malware samples we are able to apply it to. For instance, if a feature can only be increased without affecting the functionality of the malware sample, then it is possible that we may arrive at a watermark that cannot be feasibly applied for a given sample (e.g., file size can only be increased). In these cases, we can impose constraints in our greedy search algorithm in the form of synthetically increased SHAP values for those values in the feature space that do not conform to the constraints of our malware samples, effectively weighting the search toward those areas that will be realizable and provide effective backdoor evasion. we used default parameters for training LightGBM (100 trees and 31 leaves per tree). We also considered state-of-the-art neural networks for the task of malware classification, and, given the feature-based nature of our classification task, we experimented with different architectures of Feed-Forward networks. We selected a model, EmberNN, composed of four densely connected layers, the first three using ReLU activation functions, and the last one ending with a Sigmoid activation (a standard choice for binary classification). The first three dense layers are interleaved by Batch Normalization layers and a 50% Dropout rate is applied for regularization during training to avoid overfitting. Performance metrics for both clean models (before the attacks are performed) on the EMBER test set are comparable, with EmberNN performing slightly better than the publicly released LightGBM model.

**Dataset**

We evaluate our attack against a variety of machine learning models trained on widely-used malware datasets, including EMBER (Windows executables) , Contagio (PDFs) , and Drebin (Android executables) . Additionally, they have also explore the impact of various real-world constraints

**Working with dataset**

**Step-1:** Data Pre-processing

Since multiple datasets used the train test validation split is different for each dataset and not mentioned

**Step-2:** Feature Extraction (Derbin dataset)

A total 8 subsets of features are involved. S1-S4 being characteristics of the Android manifest file, and S5-S8 being extracted from the disassembled code.

Set 1- Hardware configuration

Set 2-permission requested by APK.

Set 3-Application components like service, receivers,

Set 4-APK’s communication to the OS.

Set 5-Critical API system calls which cannot be run without permissions.

Set 6-used permissions.

Set 7-API calls that access the sensitive data on smartphone.

Set 8-IP address, hostnames and URL’s extracted by the disassembled code.

**Step-3:** we test our attack on a Random Forest classifier for the PDF files, and a Linear

Support Vector Machine (SVM) classifier for the Android applications.

**Insights from the paper**

**Insight 1.** none of the mitigation approaches was able to isolate the points attacked with water marks produced with the Combined strategy on PE file.

**Insight 2.** Isolation Forest, trained on the reduced feature space, is often capable of correctly isolating all the backdoored points with relatively low false positives.

**Insight 3.** at 2% poisoning rate, the attack lowers the model accuracy on backdoored

samples to 42.9% on average We also observed minimal loss of Acc (Fb,X)

**Insight 4.** The information “barrier” that the defender does not know the attacker’s manipulation set is a fundamental one because the attacker may use adversarial malware examples that are far away from what the defender would use to train its defense model.

**Insight 5.** However, due to the strong semantic restrictions of the binaries, we cannot simply choose any arbitrary value for our backdoors.

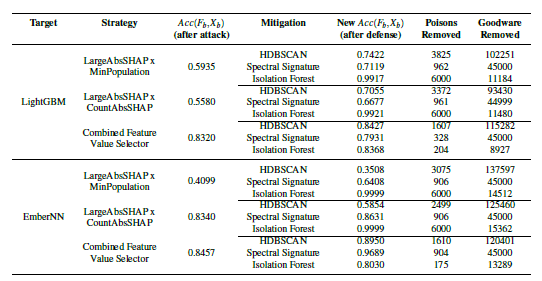
**Insight 6.** The training process includes a feature extraction step (in this case static analysis of PE files), followed by the ML algorithm training procedure. The trained malware classifiers are then deployed in the wild, and applied to new binary files to generate a label, malicious(malware) or benign (goodware).

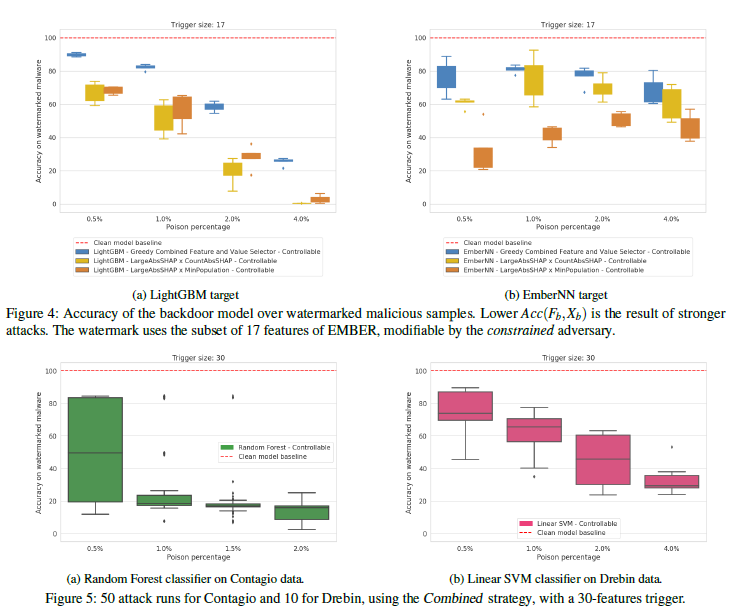
**Results and Conclusions**

We showed how to conduct backdoor poisoning attacks that are model-agnostic, do not assume control over the labeling process, and can be adapted to very restrictive adversarial

models. For instance, an attacker with the sole knowledge of the feature space can mount a realistic attack by injecting a relatively small pool of poisoned samples (1% of training set) and induce high misclassification rates in backdoored malware samples. Additionally, we designed the Combined strategy that creates backdoored points in high-density regions of the legitimate samples, making it very difficult to detect with common defenses. Based on our exploration of these attacks, we believe explanation-guided attack strategies could also be applicable to other feature-based models, outside of the security domain.

The table below shows Mitigation results for both LightGBM and EmberNN. All attacks were targeted towards the 17 controllable features, with a 1% poison set size, 6000 backdoored benign samples. We show Acc(Fb;Xb) for the backdoored model, and after the defense is applied. We also include number of poisoned and goodware points filtered out by the defensive approaches.





**[2] High accuracy android malware detection using ensemble learning**

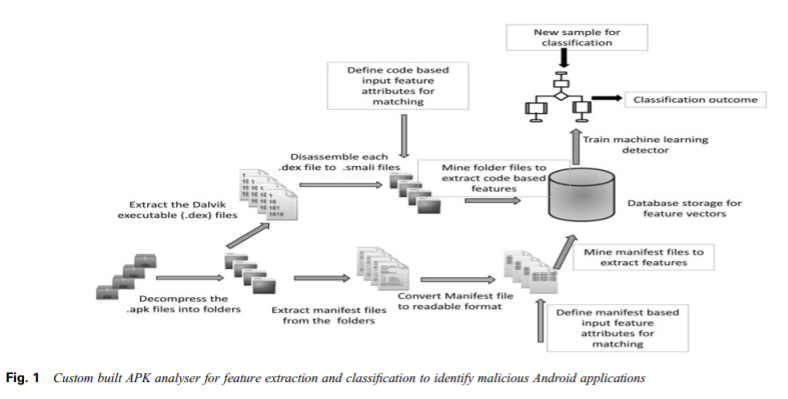
**Authors** Suleiman Y. Yerima1, Sakir Sezer, Igor Muttik

**Problem Statement**

This paper focus on the extraction of key features from Android apps using ensemble learning method to improve Machine Learning based detection framework

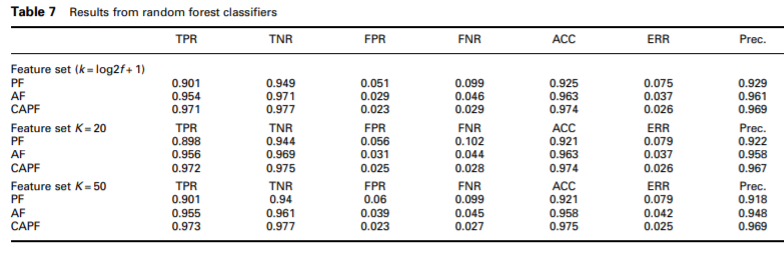
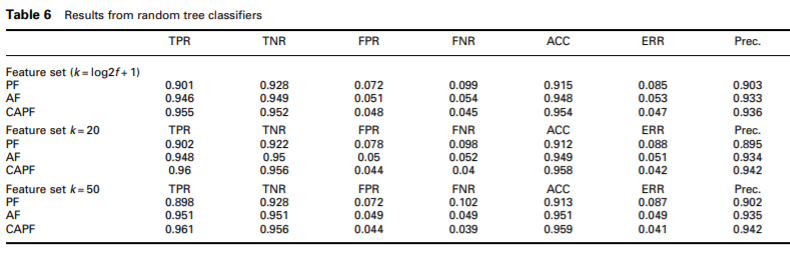
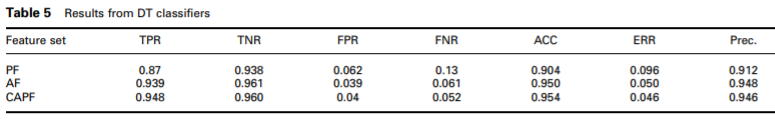
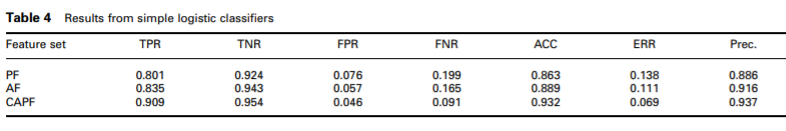
Paper mentions a framework, named AntiMalDroid, that can automatically detect whether an app is a malware or not. DroidChecker relies on the automated combination of marketplace crawlers, filtering and feature extraction, and classifiers. . Though the frameworks available we the approach followed in the paper utilised permissions and call flow graphs for training support vector machine (SVM) models to distinguish between benign and malicious Android apps.

**Algorithms/Tools/Technologies**



Tools used were AXMLPrinter.jxml.

**Models used and accuracy**



**Understandings from the paper**

The analyzing behavior, functionality, and impact of malware samples on a user system defined as malware analysis. The analyzing of malware samples variants in different ways which are signature-based, behavior-based and memory-based malware analysis.

**Malware Analysis** The process of detecting or examining the malicious code without executing it. It is a signature-based malware analysis. In static malware analysis, static features such as Metadata strings, code, and import libr aries are extracted and used in the feature selection or feature extraction phase in the machine learning classification. The input file type of static malware analysis is should be of the type exe, DLL, documents, Assembly code, byte code, etc., from these file types static features are extracted as output. The tools used for static malware analysis are PEiD, ssdeep, pafish, Yara, strings, IDA Pro, OllyDbg, LordPE, OllyDump, etc.,

**Dataset**

The data were obtained by a process that consisted to create a binary vector of permissions used for each application analyzed {1=used, 0=no used}. Moreover, the samples of malware/benign were devided by "Type"; 1 malware and 0 non-malware.

In order to investigate the effect of feature diversity, three separate feature sets were created for training models and comparative analysis: (a) feature set consisting of vectors from the 54 application attribute features only, (b) feature set of vectors from the 125 permissions only and (c) feature set consisting of vectors from a mix of all the (diverse) 179 property vectors. The learning algorithms were investigated with the three feature sets in order to evaluate their classification performance using the Accuracy and error rate.

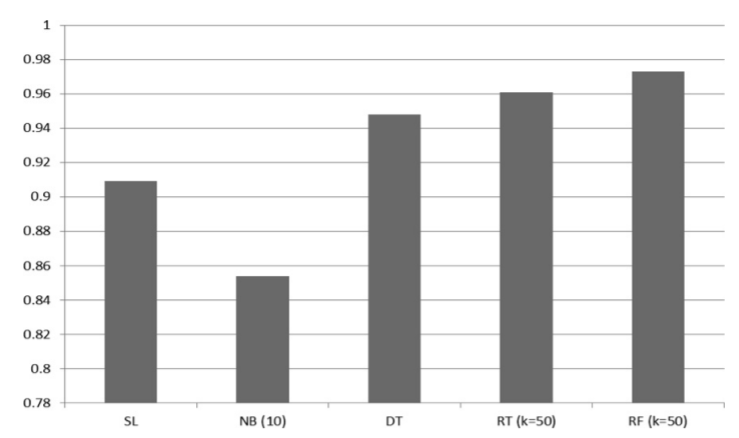
**Working with dataset**

**Step-1:** The APK analyser was applied to a total of 6863 applications (obtained from McAfee’s internal repository); 2925 of these were malware while 3938 apps were benign

**Step-2:** The extracted features were converted into binary feature vectors with a 0 or 1 indicating the absence/presence of a feature.

**Step-3:** Out of the initial 65 non-permission based features 54 of these were selected to produce feature vectors for the training phase.

**Step-4:** After applying the classification models, performance of each model is evaluated using True Positive Rates, False Positive rates, True Negative rates, False Negative Rates, Accuracy.



**Results and Conclusions**

This paper presented a new ensemble learning-based Android malware detection approach which can effectively improve detection rates to 97–99% with low false positives by harnessing large mixed feature sets in ways infeasible with traditional machine learning. With this approach, there is no requirement for feature selection step to eliminate ‘less relevant’ features. This use of an extensive mixed feature set provides robustness and resilience to code obfuscation and other anti-analysis techniques being employed by malware authors vastly improving the chances of prompt zero-day malware detection. Experiments performed with large malware dataset from a leading AV vendor demonstrate the effectiveness of the proposed scheme and the higher fidelity achievable compared to traditional approaches.

**[3] Feature Extraction using Hybrid Analysis for Android Malware Detection Framework**

**Authors** Soe Myint Myat, Myanmar May Thu Kyaw

**Problem Statement**

This paper focus on the extraction of key features from Android apps using hybrid analysis method to improve Machine Learning based detection framework

Paper mentions a framework, named ADetect (Android Detector), that can automatically detect whether an app is a malware or not. ADetect relies on the automated combination of marketplace crawlers, filtering and feature extraction, and classifiers. It is meant to process large quantities of applications, filter out applications which are either clean or known malware. This paper focuses on the feature extraction for malware detection. We propose a hybrid security solution, integrated static and dynamic analysis method, to analyses and characterize an unknown executable file.

**Understandings from the paper**

It provides a detecting architecture aiming at identifying harmful Android applications without modifying the Android firmware. A hybrid feature selection method was proposed by addressing the selecting of key features from Android apps. Three Machine Learning classifiers is used to evaluate malware classification accuracy in our feature set. According to the result, the approach can be considered as an effective approach in malware detection. However, the time is too long in real smart phone because of hardware requirement. A major benefit of the approach is that the system is designed as platform-independent so that smart devices with different versions of Android OS can use it.As a future work, it can be designed to develop a real-time malware detection infrastructure.

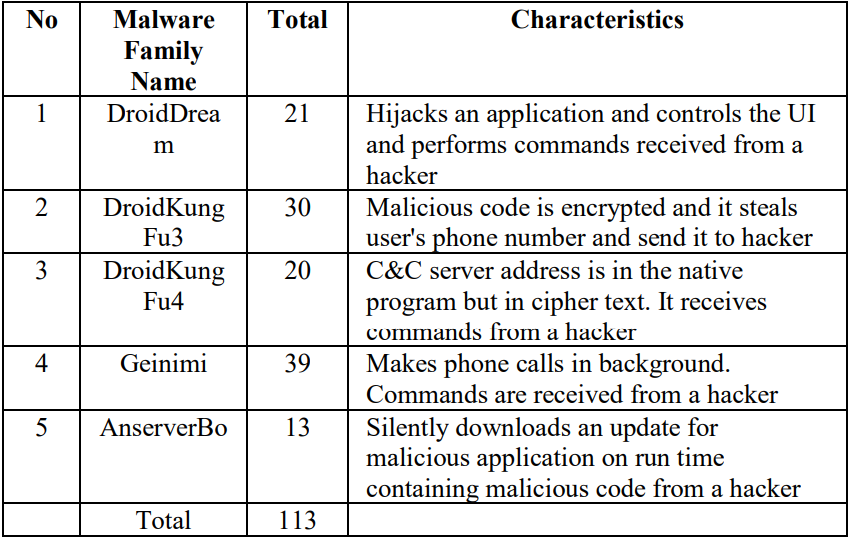
Some of the benefits of feature selection are as follows:

* Feature selection makes it possible to reduce dimensionality of datasets because with less data, it is possible to easily visualize the trend in data.
* Analyzing datasets involve processing vast amount of data and therefore, reducing them to a useful subset not only saves the time and cost of experiments, but also minimizes the time for real world implementation.
* Feature selection removes noisy and irrelevant data from datasets leading to more accurate results of machine learning algorithms.

**Dataset and its Working**

**Step-1:** Data Collection

In this data collection phase, we use two main approaches to collect the data. Firstly, we crawl malware samples directly from well-known Android malware blogs such as Contagio Mobile Malware Mini Dump. Because of no standard dataset for benign application, we collected dataset from Google Play Store which is considered as the official market with the least possibility of malware application. We have collected total 219 applications from various sources. In total, 7 static and dynamic feature sets are extracted from our malicious and benign Android applications including used permissions, requested permissions, permission request APIs, network APIs, suspicious calls, and more.



**Step-2: Hybrid Analysis Features**

**Used permissions:** Some of Android applications request multiple permissions. However, they use only a subset of the requested permissions. We can get more exact observation of apps intension by extracting the used permissions.

**Requested permissions:** Permission is one of the most important security mechanisms introduced by Android. Most user normally grant the permissions without the knowledge, therefore an application can install itself and can perform malicious behaviors.

**Permission request APIs:** API calls can be requested Android permission. For example, a sendDataMessage call requests permission SEND SMS and to receive SMS, developer use android.permission.RECEIVE\_SMS.

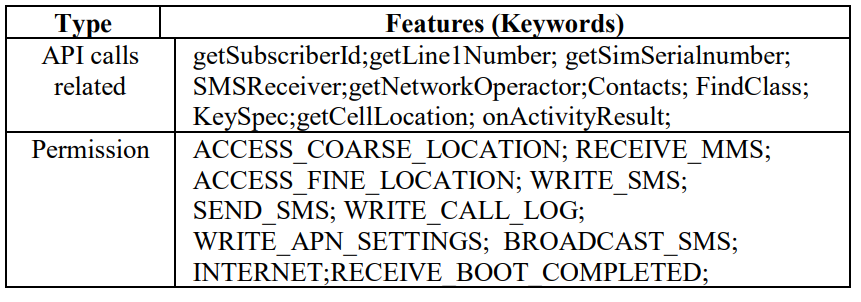
**Network APIs:** Malwares are now try to access the network and then send out sensitive data by using network APIs.

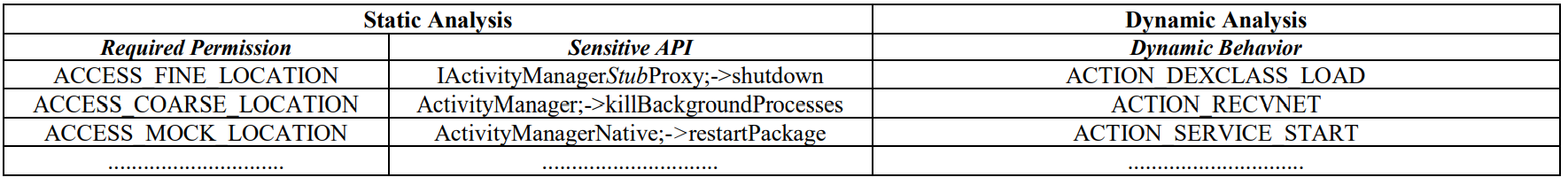
**Suspicious calls:** These Suspicious API calls such as communicating over the network, sending and receiving messages, and executing external commands are frequently used by malware developers.

**Providers:** The provider is a subset of component that support structured access to data managed by the application. All content providers in application must be defined in a element in the manifest file; otherwise, the system is unaware of them and doesn't run them.

**Step-3:** **Feature Extraction and Selection**

Android apps are packed into apk format, and the features we are interested in are encrypted, such as permissions, APIs, actions, IP and URLs. To systematically characterize, we combine static and dynamic analysis to extract interesting features. All features fall under one of three types: dynamic behaviors, required permissions and sensitive APIs. Among them, dynamic behaviors are extracted with dynamic analysis, whereas sensitive APIs and required permissions are extracted with static analysis.



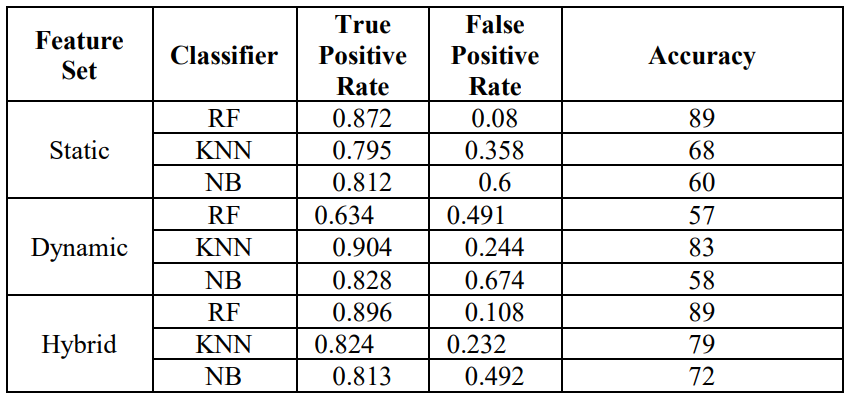


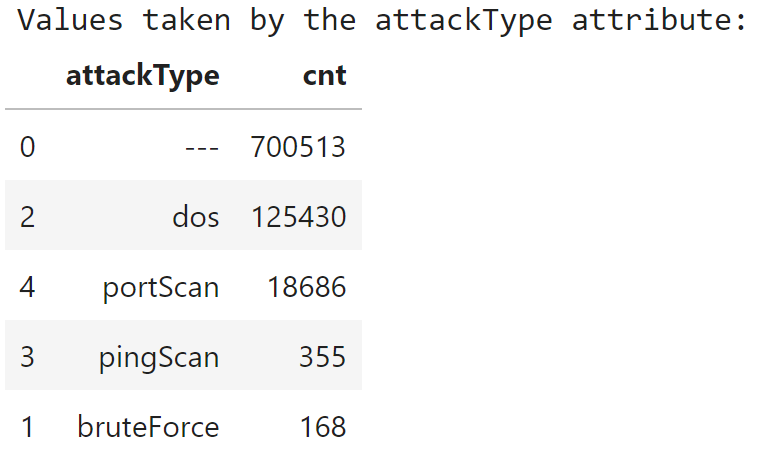
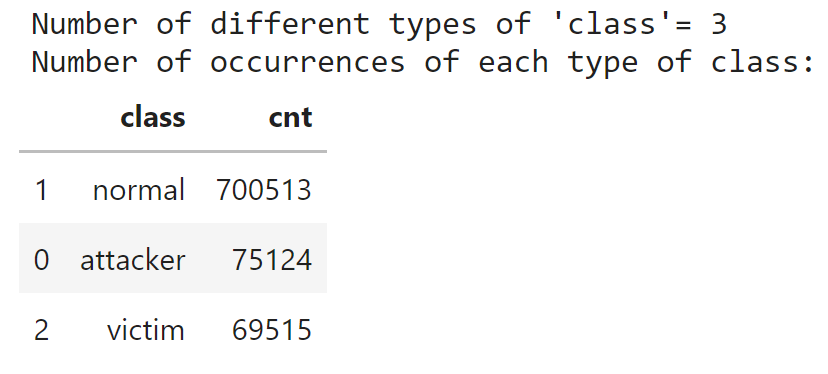
**Step-4: Machine Learning based Classifier**

Traditional machine learning models (e.g., SVM and C4.5) that have less than three layers of computation units are considered to have shallow architectures. In practical use, a deep learning model can be constructed with different deep architectures, e.g., Deep Belief Networks (DBN) and convolutional neural networks. We chose DBN architecture to construct our deep learning model and characterize Android apps. For our proposed framework, the construction of a deep learning model has two phases, the unsupervised pretraining phase and supervised back-propagation phases. In the pre-training phase, the DBN is hierarchically built by stacking a number of Restricted Boltzmann Machines (RBM), with the deep neural network regarded as a latent variable model, which is beneficial for gradually evolving high-level representations. In the back-propagation phase, the pre-trained DBN is finetuned with labeled samples in a supervised manner. The deep learning model uses the same app set in both phases of the training process. In this way, the deep learning model will be completely built.

**Results and Conclusions**

Three cases are evaluated in this section. Firstly, static analysis was carried out using only the permission data for training and testing. Secondly, dynamic analysis (system call frequency data) was analyzed for training and testing. Lastly, training and testing was carried out by combining static and dynamic such as the permission data and system calls frequency data. Accuracy of a test is evaluated on how well the test is able to distinguish between a malware and benign. We use three classifiers Random Forest (RF), K-nearest Neighbour (KNN) and Naïve Bayes (NB) to evaluate our hybrid feature selection method. The evaluation was performed by measuring true positive rate, false positive rate, precision, recall, F-measure and accuracy.





The differences and similarities between the Train and the Test datasets (respectively weeks 1 and 2):

The obvious difference in the two datasets is the number of rows, which goes from 845,152 for week1 to 1,031,073 for week2.

In the first week, the pingScan attack type is more frequent than the brute force (which is the least frequent of all attack types). In the second week, it is the other way around. For the two datasets, Denials of Service represent the most frequent attack type.

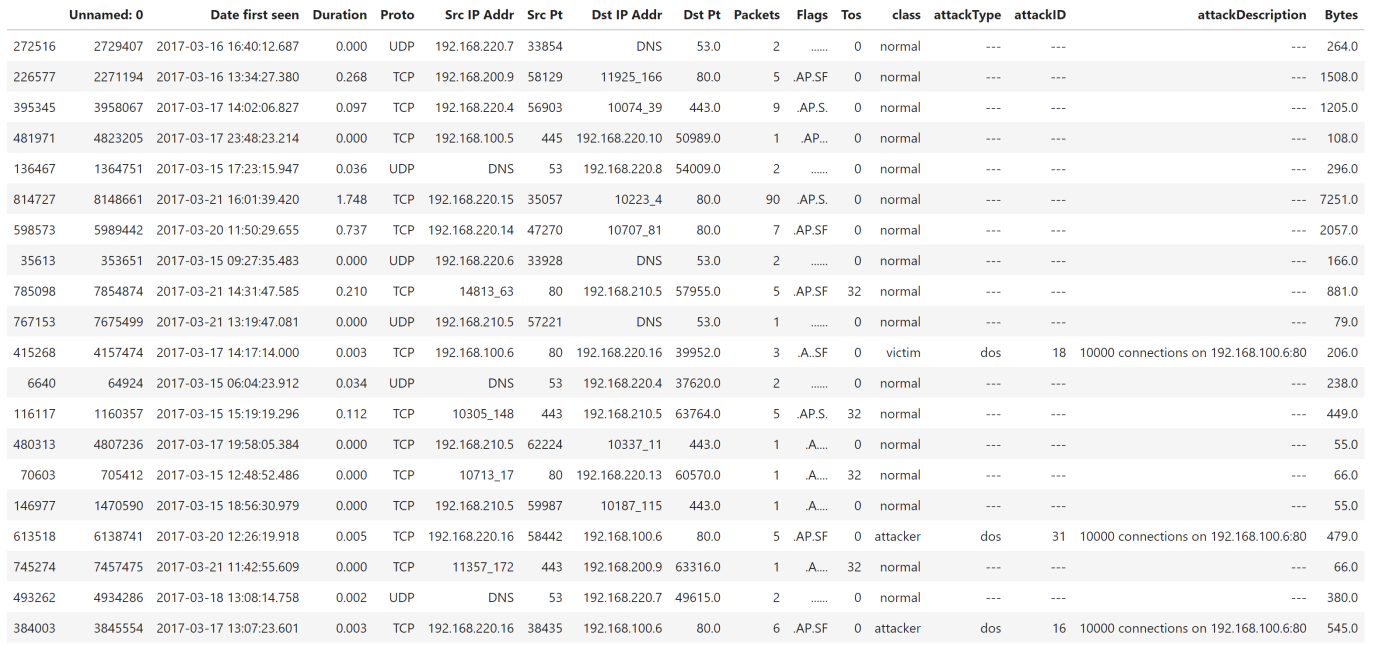
Our three numeric variables - the columns 'Bytes', 'Duration' and 'Packets' - are almost identical statistically in both datasets. They all have a high range of values, but a relatively very low median, which tells us that they contain abnormally high values.

It is only when it comes to time-related analysis that we truly see a difference between the two weeks. Indeed, the proportion of traffic flows that happened on Thursday and Friday is a lot higher during the second week than the first one. Also, whereas attacks happened on every single day of the working week during the first week, they happened only on Thursday and Friday during the second.

During the second week, a negligeable amount of attacks happened before 10am. A bigger proportion of those happened during week1. Individually, network flows had a 31.5% chance of being an attack on Friday during the first week. That percentage is slightly higher for week two: 39%. The difference in the individual probabilities of being an attack for network flows was a lot more notable when looking at the hours of the day. Whereas these probabilities almost looked like a bell curve peaking at 2pm in the first week, they were more like regular spikes every other hour, starting from 11am and ending at 6pm.

Here we define two datasets:

* train\_df : it contains the first week data
* test\_df : it contains the second week data
* "Date first seen" is of type 'object'. It needs to be changed to datetime, so as to extract the day of the week and hour of the day.
* "Bytes" is a numerical feature, but here it is of type 'object', so we need to change it to a float to use it in the classification.



#### **Initial Findings-1**

Logistic regression performs badly, which is to be expected given that the prediction target is non-binary. Its accuracy is actually not bad, but that's because it arbitrarly assigns almost all the observations to the most likely/popolous class. This means that the prediction is actually useless, because our aim is most likely to separate normal behavior from anomalous behavior, and we are not achieving that. It is interesting to see what will happen when considering more features, in that case logistic regression might work better.

Decision Tree and Random Forest perform very well, they have a super high accuracy and their confusion matrix looks like it is actually predicting the anomalies. They can be optimized further by adding categorical features, but we need to be careful about overfitting. Because their performance is good on the test set, we have no reason to believe it is overfitting too badly as of now.

SVM is definitely not the right algorithm to analyze this dataset. It takes over 5 hours to run and yields a below average result. That's why we implemented a bagged version of the algorithm, that takes only a portion of the dataset at the time to fit the model, and repeats the process a few times, to return the average findings. Its performance is comparable to the non bagged version of the algorithm, so we decide to continue using the bagged version. Its performance is not good though, it always assign the observations to the most popolous class.

Neural Network performs very well. It is not the best performer here, but we believe that with further adjustments of the layering and parameters it could outperform Random Forest.

#### **Initial Findings-2**

We have the results we were expecting, but with a few additional interesting findings.

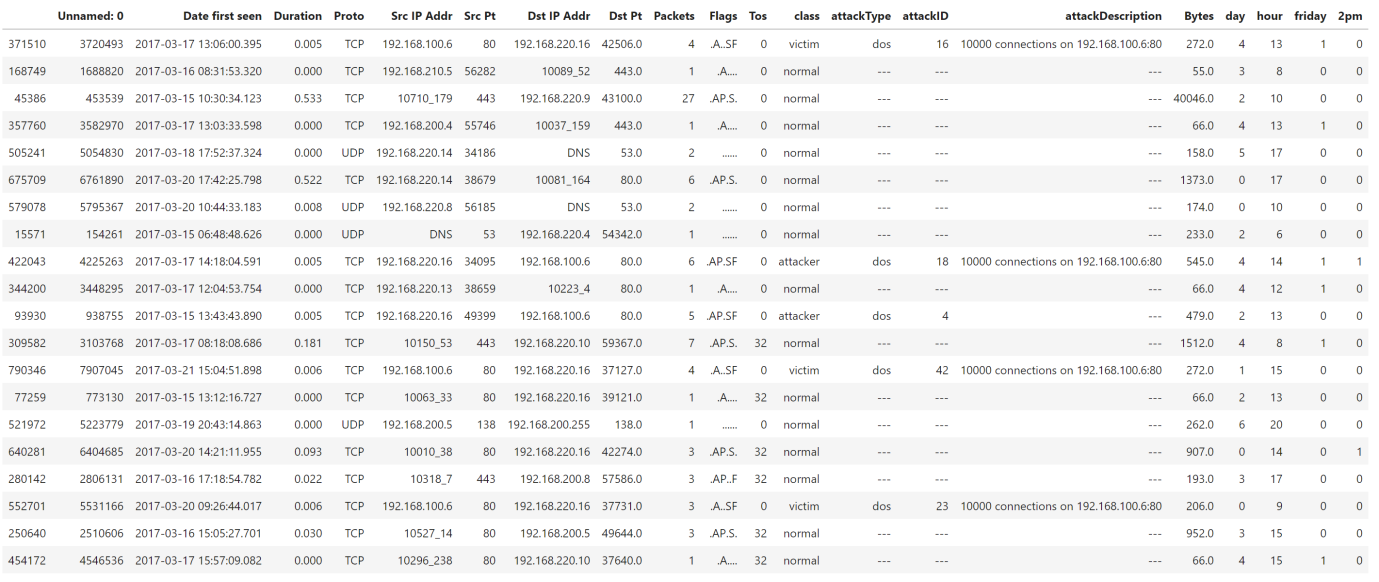
Logistic Regression performs poorly, just as with the classifications above.

We can really see the difference in performance between Decision Tree and Random Forest thank to this analysis. Decision tree still performs well, but struggles predicting the under represented Attack types. Random Forest instead can predict the under represented categories very well. In this scenario we would choose for sure Random Forest as our top performer.

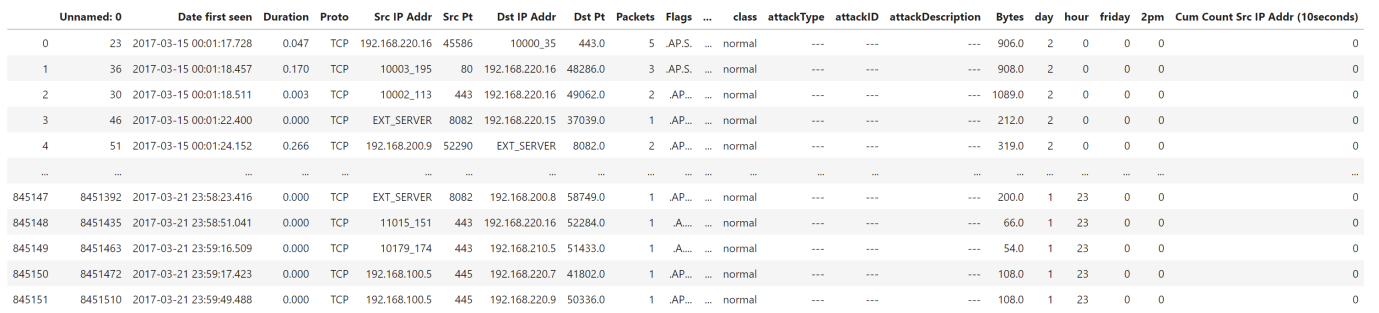
Naive Bayes keeps performing poorly.

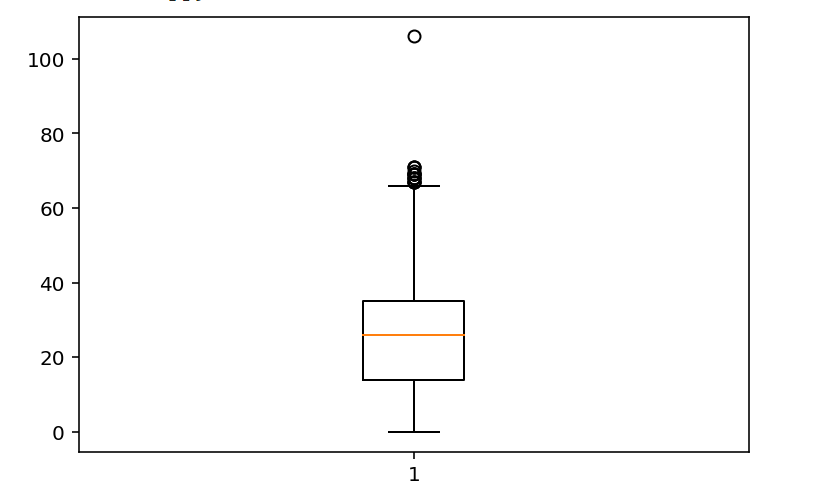
BaggingSVM yields an error. This error most likely occurs because of the highly unbalanced dataset. Since the BaggingSVM takes randomly 1% of the data available it is possible that the underrperesented values are not in the sample. Hence, we get a shape error.

Neural Network struggles in this scenario, it never predicts the under represented categories.



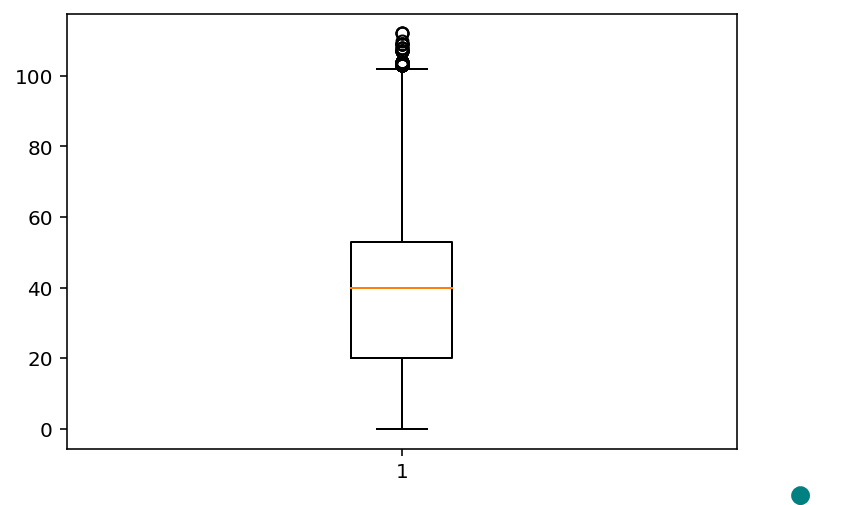
## Counting total requests in a given amount of seconds





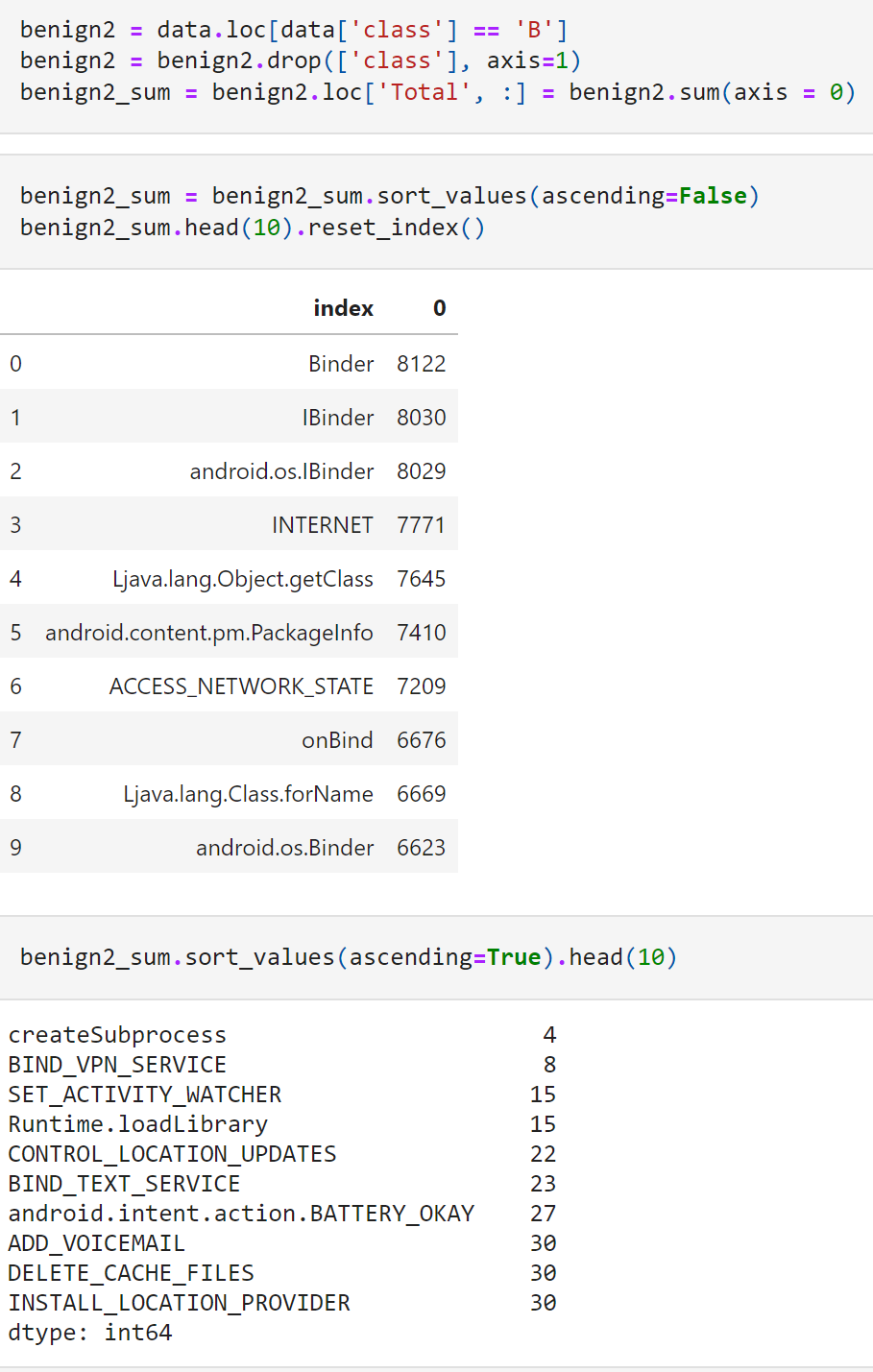
It is interesting to note that some applications have been categorised as malware despite acting like none of our recorded behaviours.

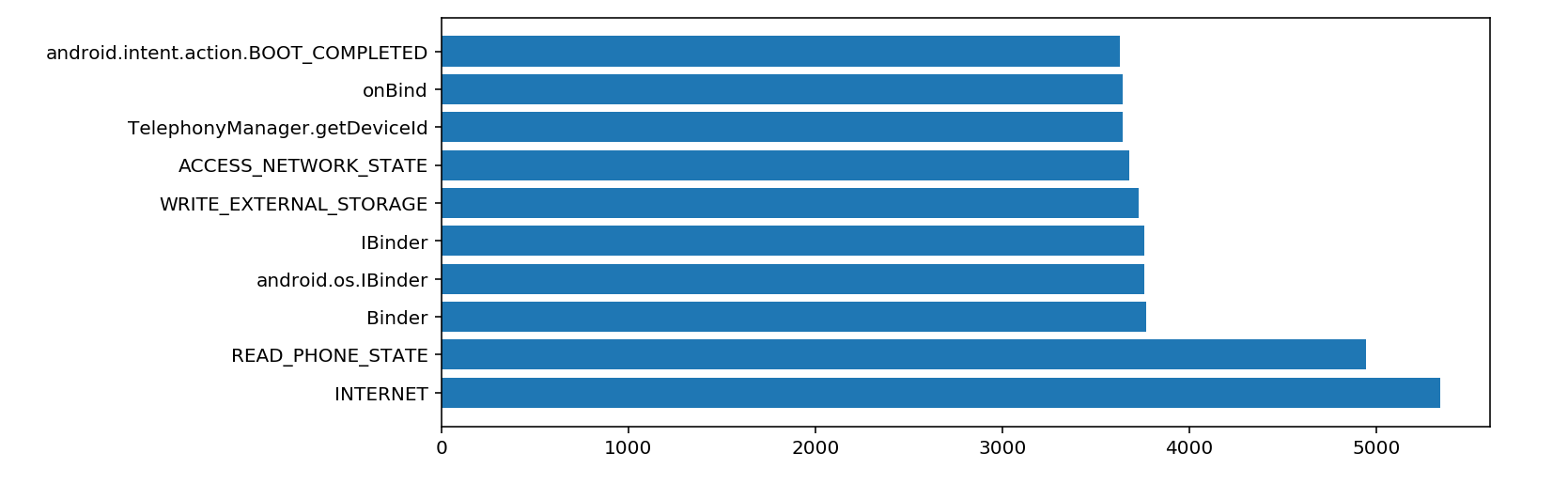
The biggest amount of behaviours a malware has shown is 106. We can therefore say that the amount of different behaviours of malware has a high range.



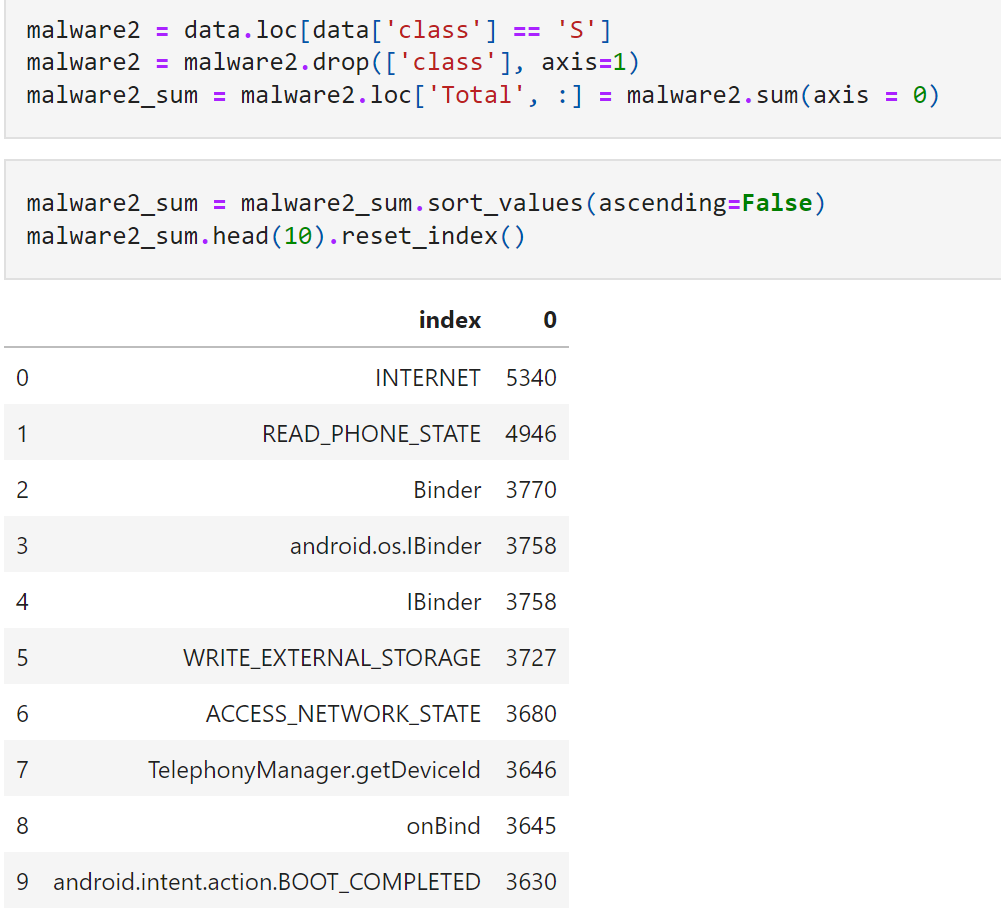
The biggest amount of behaviours a benign application has shown is 112, which is higher than the maximum amount for a malware (106). Therefore, we can conclude that the number of behaviours is not predictive of the class of an application. On average the benign applications act according to a bigger variety of behaviours than the malware.

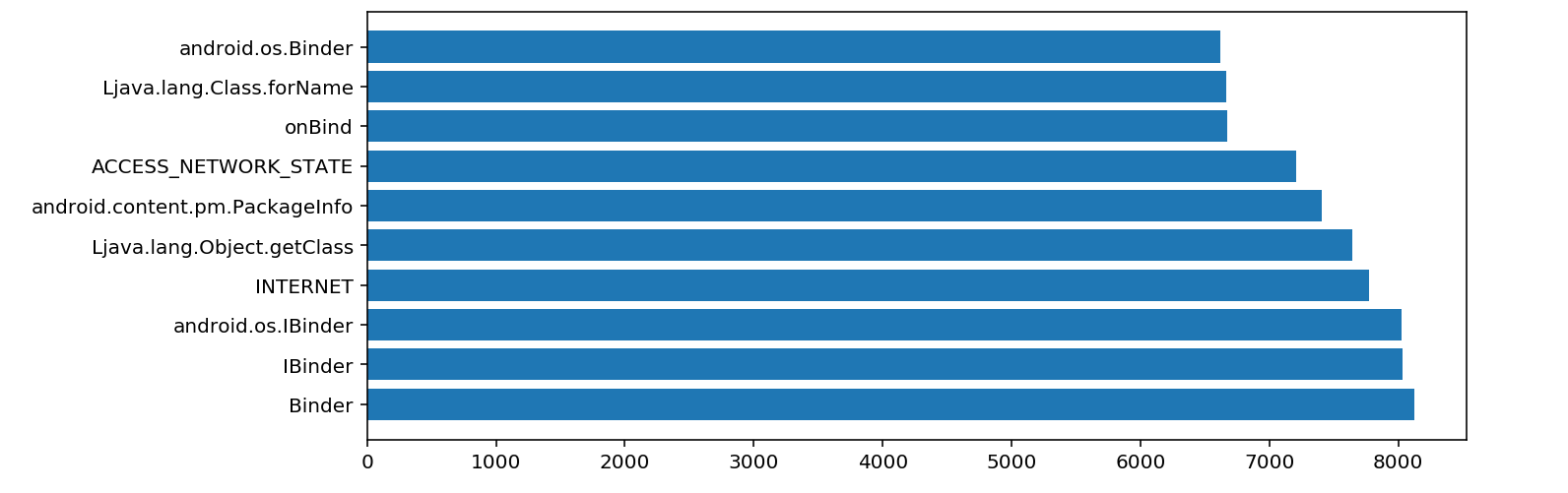
**Benign Data**





**Malicious Data**





The two graphs show that the 10 most common behaviours differ slightly from benign to malware applications. The most common one for malware is 'INTERNET', and 'Binder' for benign apps.

The general results of the classifiers on the Android malware dataset in regard to predicting the class attirbute is very good. Most notably, the random forest algorithm had the best accuracy performance on the second week test data which underlines its capability to generalize effectively. This result is further supported by confusion-matrix which shows that the algorithm only mistakenly classified in 5 instances class 0 as class 1 and 37 times class 1 as class 0.

# **Additional work**

# **A Brief Introduction to The Microsoft Malware Detection Case Study**

## 1. Business/Real-world Problem

### 1.1 What is Malware?

The term malware is a contraction of malicious software. Put simply, malware is any piece of software that was written with the intent of doing harm to data, devices or to people.

Source: <https://www.avg.com/en/signal/what-is-malware>

### 1.2 Problem Statement

In the past few years, the malware industry has grown very rapidly that, the syndicates invest heavily in technologies to evade traditional protection, forcing the anti-malware groups/communities to build more robust softwares to detect and terminate these attacks. The major part of protecting a computer system from a malware attack is to **identify whether a given piece of file/software is a malware.**

### 1.3 Dataset Overview

Microsoft has been very active in building anti-malware products over the years and it runs it’s anti-malware utilities over 150 million computers around the world. This generates tens of millions of daily data points to be analyzed as potential malware. In order to be effective in analyzing and classifying such large amounts of data, we need to be able to group them into groups and identify their respective families. This dataset provided by Microsoft contains about 9 classes of malware. , Source: <https://www.kaggle.com/c/malware-classification>

### 1.4 Real-world/Business objectives and constraints.

1. Minimize multi-class error.
2. Multi-class probability estimates.
3. Malware detection should not take hours and block the user's computer. It should fininsh in a few seconds or a minute.

## 2. Data Overview

Source: <https://www.kaggle.com/c/malware-classification/data>

For every malware, we have two files

* 1. .asm file (read more: <https://www.reviversoft.com/file-extensions/asm>)
  2. .bytes file (the raw data contains the hexadecimal representation of the file's binary content, without the PE header)

Total train dataset consist of 200GB data out of which 50Gb of data is .bytes files and 150GB of data is .asm files: Lots of Data for a single-box/computer.

There are total 10,868 .bytes files and 10,868 asm files total 21,736 files.

There are 9 types of malwares (9 classes) in our given data

* 1. Ramnit
  2. Lollipop
  3. Kelihos\_ver3
  4. Vundo
  5. Simda
  6. Tracur
  7. Kelihos\_ver1
  8. Obfuscator.ACY
  9. Gatak

For each file, the raw data contains the hexadecimal representation of the file's binary content, without the PE header (to ensure sterility). You are also provided a metadata manifest, which is a log containing various metadata information extracted from the binary, such as function calls, strings, etc. This was generated using the IDA disassembler tool. Your task is to develop the best mechanism for classifying files in the test set into their respective family affiliations.

## 3. High Level Steps to Run Case Study

1. Data overview

2. Mapping the real world problem to ml problem

3. Data Preprocessing and Exploratory Data Analysis(EDA)

4. Train,Test and CV split

5. Modeling

6. Prediction

## 4. Major Challenges

1. Disk Space (This case study requires approx. 250 GB free space on your disk)

2. Time (Approx. 48 hours to fetch Features from asm files (150 GB)), Please use data/asmoutputfile.csv file insteadof processing whole asm files.

3. Feature Engineering (Image Based Features Extraction from asm files)

**5. ML Models**

5.1 ML Models on .byte files unigram

* Random Model
* K Nearest Neighbour Classification
* Logistic Regression
* Random Forest Classifier
* XgBoost Classification
* XgBoost Classification with best hyper parameters using RandomSearch

5.2 ML Models on .asm files unigram

* K Nearest Neighbour Classification
* Logistic Regression
* Random Forest Classifier
* XgBoost Classification
* XgBoost Classification with best hyper parameters using RandomSearch

5.3 Select Best ML Models

* Random Forest Classifier
* XgBoost Classification

5.4 ML Models on .byte + .asm files unigrams

* Random Forest Classifier
* XgBoost Classification
* XgBoost Classification with best hyper parameters using RandomSearch

5.5 ML Models on .byte files bigram

* Random Forest Classifier
* XgBoost Classification
* XgBoost Classification with best hyper parameters using RandomSearch

5.6 ML Models on Image Based Features Extraction from asm files

* Random Forest Classifier
* XgBoost Classification
* XgBoost Classification with best hyper parameters using RandomSearch

5.7 I tried following combination of features with best models (Random Forest and XgBoost)

* ASM image features + bytes uni-gram features
* ASM unigram + ASM extracted image features
* ASM unigram + ASM image features + ByteFile unigram

**6. Result**

ASM unigram + ASM image features + ByteFile unigram (XgBoost Classifier With Best HyperParameter) reduced the log-loss < 0.01 (0.005282303659991528)

