A thesis submitted in partial satisfaction of the requirements
for the degree of Master of Computer Science and Engineering
in the Graduate School of the University of Aizu

Detecting Android malware applications using Static Analysis and Machine learning

by

Cristian Naoki Kamia

March 2020

The thesis titled

Detecting Android malware applications using Static Analysis and Machine learning

by

Cristian Naoki Kamia

is reviewed and approved by:

Main referee
Senior Associate Professor
Vitaly Klyuev
Professor
Pyshkin
Professor
Paik

THE UNIVERSITY OF AIZU

March 2020

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Programming Interface
APK	Android Package
ART	Android Runtime
CSV	Comma-separated values
HAL	Hardware Application Layer

Abstract

Chapter 1

Introduction

The android operating system, which is provided by Google, is predicted to continue to have a dramatic increase in the market with around 1.5 billion Android-based devices to be shipped by 2021 sta. It is currently leading the mobile OS market with over 80% market share compared to iOS, Windows, Blackberry, and Symbian OS as Figure 1.1. The availability of diverse Android markets such as Google Play, the official store, and third-party markets makes Android devices a popular target to not only legitimate developers, but also malware developers. Over one billion devices have been sold and more than 65 billion downloads have been made from Google Play. Android apps can be found in different categories, such as educational apps, gaming apps, social media apps, entertainment apps, banking apps, etc.

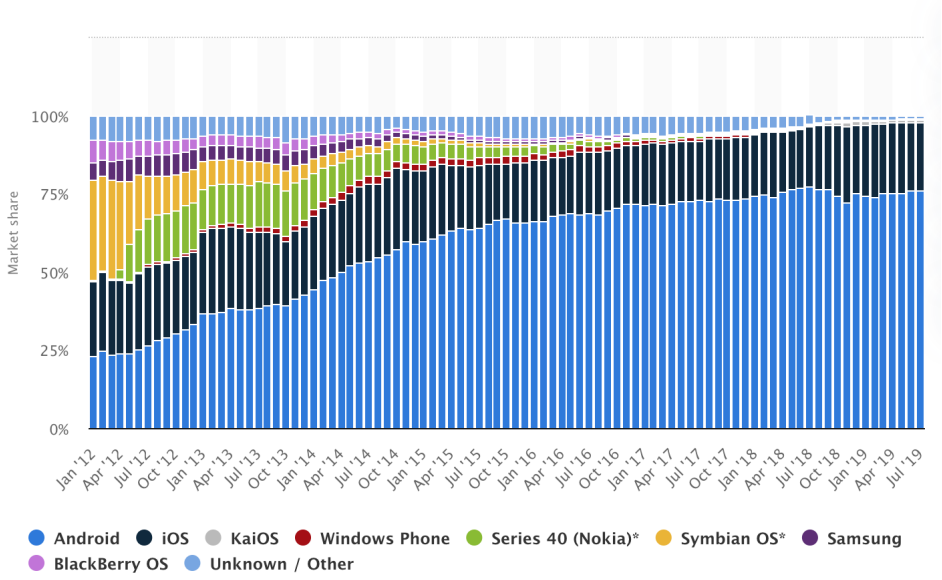

Figure 1.1: Mobile operating systems's market share worldwide from January 2012 to July 2019 [1]

As a technology that is open source and widely adopted, Android is facing many challenges especially with malicious applications. The malware infected apps have the ability to send text messages to premium rate numbers without the user acknowledgment, gain access to private data, or even install code that can download and execute additional malware on the victims device. The malware can also be used to create mobile botnets [4]. Over the last few years, the number of malware samples attacking Android has significantly increased as you can see at Figure 1.2. Attacks on other connected things around the house gained momentum as well. While hidden apps and Adware remain by far the most common form of mobile threats in

Android, the others are growing and learning how to infect other types of devices as well. According to a recent report from McAfee [2], detections of backdoors,cryptomining, fake apps, and banking Trojans all increased substantially in the latter half of the year.

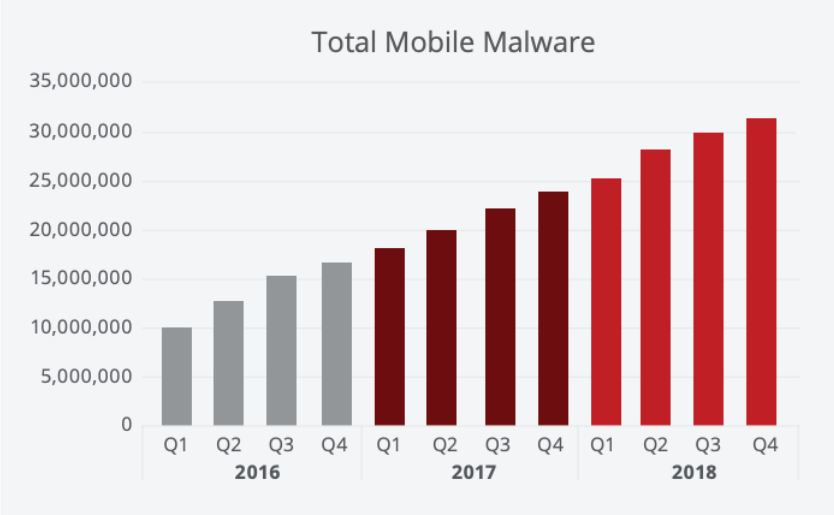

Figure 1.2: Total mobile malware [2]

In order to mitigate the spread of malware, Google introduced a detection mechanism to its app market in Feb 2012 called Bouncer. Bouncer tests submitted applications in a sandbox for five minutes in order to detect any harmful behaviors; however, it has been shown that bouncer can be evaded by means of some simple detection avoidance methods. Alongside Bouncer, Google introduced Google Play protect in the Google 2017 event [5]. Google Play Protect is an always-on service that scans the applications automatically even after installation to ensure that the installed applications remains harmless 24/7. It has been reported that over 50 billion apps are scanned and verified every day regardless of where they were download from. However, according to McAfee, Google Play Protect also failed when tested against malware discovered in the previous 90 days in 2017 [6]. Furthermore, most third-party stores do not have the capability to scan and detect submitted harmful applications. Clearly, there is still a need for additional research into efficient methods to detect zero-day Android malware in the wild in order to overcome the aforementioned challenges.

Cyber attacks manage to produce unprecedented levels of disruption, where attackers usually leverage diverse tools and tactics, such as zero-day vulnerabilities and malware. This situation makes malware detection techniques worth studying and improving, in order to prevent and/or mitigate the effects of cyber attacks. Machine learning techniques can help to satisfy this demand,building classifiers that discern whether a precise Android application is malware or benignware. Algorithms such as Decision Trees, Support-Vector Machines and NaiveBayes, to name a few, are able to build such classifiers.Going further, ensemble methods for machine learning aim at effectively integrating many kinds of classification methods and learners to benefit from each ones advantages and overcome their individual drawbacks, hence improving the overall performance of the classication.

Various approaches have been proposed in previous works with the intention of detecting Android malware. These approaches are categorized into static analysis, dynamic analysis or hybrid analysis (where static and dynamic are used together). The methods based on static analysis reverse engineers the application for malicious code analysis. Arp et al. (2014) [7], M. Yusof et al [8]. Fan et al. (2017) [9], are few examples of detection methods using static analysis. Static analysis of Android malware can rely on Java bytecode extracted by disassembling an application. The manifest file is also a source of information for static analysis. By contrast, dynamic analysis executes the application in a controlled environment such as an emulator, or a

real device with the purpose of tracing its behavior. Several dynamic approaches, one of them is Alzaylaee, M.K., Yerima, S. Y., and Sezer S. (2016) DroidBox [3]. However, the efficiency of these approaches rely on the ability to detect the malicious behavior during the runtime while providing the perfect environment to kick-start the malicious code. And many approaches do not have an end-to-end system to detect malware. They have some separated steps to predict and are not easy to use for users.

In this paper, we will focus on Static analysis methods to detect malware in APK using features extracted from these APK files with reverse engineering methods. We used a tool named AndroPytool [3] to extract 7 different Static features described at table1.1.

Table 1.1: Static features used in our proposed system

	Feature name	**Description**
Static Feature	API calls	Count of system calls performed by an APK
	Opcodes	Count of opcodes performed by an APK
	Permissions	Which permissions uses the APK
	Intent receivers	Set of an APKs receivers
	Intent services	Services used by an application
	Intent activities	Activities declared by an APK
	System commands	Set of system commands ran by the app

Using these features, we trained several machine learning models to predict if the application is a malware or not. Then we use these models predictions to train other new models to predict the malware application. This technique is called Stacking. It is a way to ensemble multiple classifications or regression models. This method is often used by data scientists in Kaggle [10] competitions to improve the final prediction and is one of the state-of-art techniques in table data competitions. Furthermore, we propose an end-to-end system that uses for input an android APK and output the result of detection, benign application or malware application.

Chapter 2

Background

2.1 Overview of Android OS

The Android architecture is structured in a series of layers (see 2.1) offering different components and functionalities at different levels of abstraction, from hardware to system applications. The bottom layer is composed by the Linux Kernel. This is the core system of Android, on top of which all the components and layers are deployed. It also manages some of the most important security related policies. The next layer, in ascending order, is the Hardware Application Layer (HAL), in charge of allowing access to the hardware from components in upper layers. It contains independent modules for each hardware component.

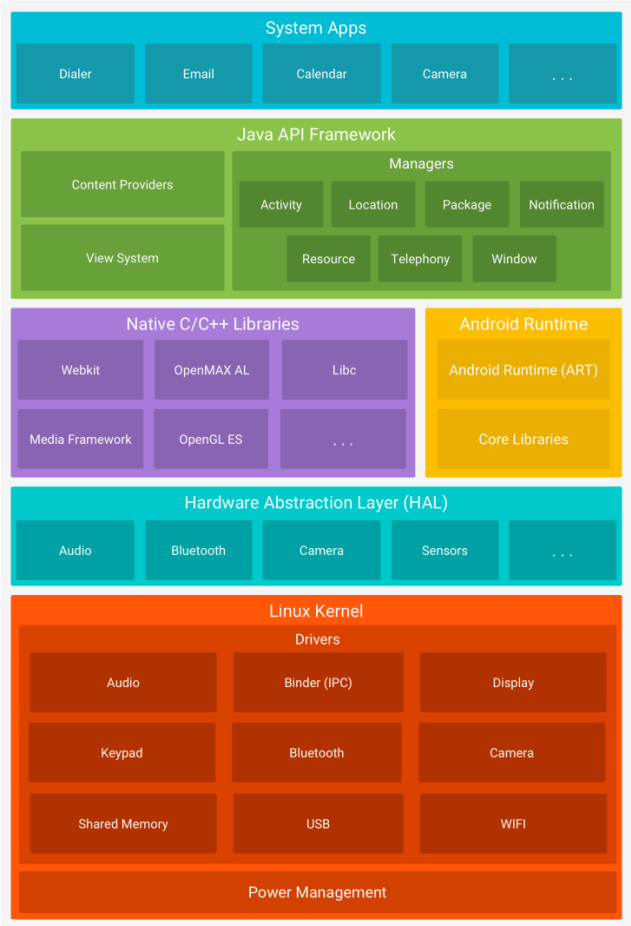

Figure 2.1: Android operating system architecture

Two layers are placed above. The first one, the Native C/C++ Libraries, used by developers who build their applications in any of these programming languages or also employed by the Java Application Programming Interface (API). The second one is the Android Runtime (ART), which has replaced Dalvik. This environment allows running multiple virtual machines where applications developed in Java execute in isolation.

On top of the two previously described parallel layers, it is possible to find the Java API framework. Through this API, it is possible to access all functionalities offered by the Android Operating system. The main objective of this layer is to facilitate the process of developing new applications or reusing code in a rich environment designed to help the developer. Finally, the top layer contains a set of system applications. They provide basic functionalities to the user, such as internet browser, phone, email or calendar, but they can be replaced by other applications installed by the user.

In Android, applications are distributed in files with extension .apk, which stands for APK. These compressed files in zip format contain the necessary code, data, and resources required to execute the application.

2.2 APK file structure

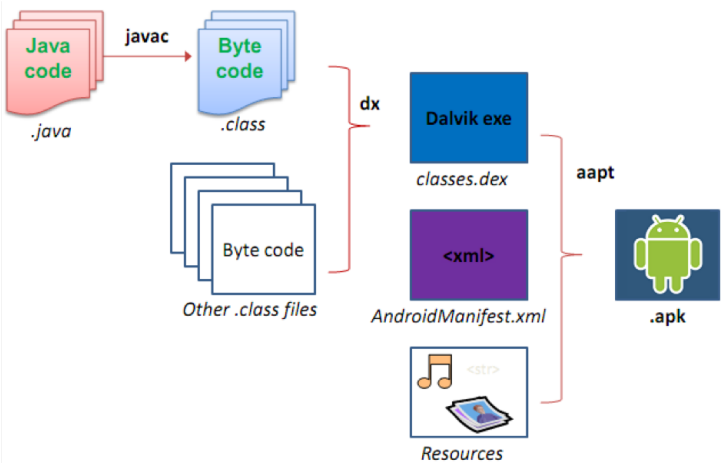

Figure 2.2: Overview of the different files and folders obtained after unzipping an APK

Figure2.2 presents an overview of the different files and folders obtained after unzipping an APK. However, since some of these files contain encrypted information, it is required to use specific tools such as AndroGuard [11] or Apktool [12] to extract the human readable version of each file. The files contained in the different folders provide varied information which can be used to categorize the behavior of the sample. For instance, /META-INF/ includes certificates, developer information or information to run the jar file. /assets/, resources.arsc and /res/ are related to different mechanisms to import resources. The /lib/ folder stores compiled libraries. Finally, two files provide the most relevant features when facing a malware analysis task and which are shown in Figure2.1. classes.dex defines the code of the application in the form of Dalvik bytecode. From here, a list of API calls, system commands or receivers defined can be retrieved. The second important file is the AndroidManifest.xml, which declares a list of permissions, the package name or a relation of intent filters.

2.3 Android Malware

Android malware applications primarily consist of Trojans. A typical Android Trojans might trick the user by using icons or user interfaces that mimic a benign application. Android Trojans

often display a service level agreement during installation which obtains permissions to access a users personal information, such as the phone number. The Trojan can then, for example, send SMS to premium rate numbers in the background. Android Trojans are also often used as spyware. Such malicious applications can gain access to a users private information and send it to a private server. The main purpose of such spyware is to steal information such as phone location, bank or credit card details, passwords, text messages, contacts, online browsing activity, and so on. A more sophisticated implementation might also include botnet capabilities.

2.4 Analysis techniques for applications

2.4.1 Static Analysis

In the static analysis, the analysis of the applications is done and the features are extracted without executing the application on an emulator or device. In comparison to other analysis techniques for android malware detection, static analysis consumes fewer resources and time as it does not involves execution of the application. The major disadvantage of this analysis is code obfuscation because of which detecting the malicious behavior of the application becomes difficult as pattern matching is not possible. This analysis can detect runtime errors, logical inconsistencies, and possible security violations.

Also static analysis includes the use of reverse engineering techniques in order to access the set of instructions that define the application operation. Furthermore, and focused on the Android platform, a large variety of characteristics can be revealed using this kind of analysis. Information gathered from the Android Manifest or from the resources are included into this category.

2.4.2 Dynamic Analysis

Dynamic analysis is the testing and evaluation of a program by executing data in real-time. The objective is to find errors in a program while it is running, rather than by repeatedly examining the code offline. It is a detection technique which aims at evaluating malware by executing the application and the main advantage of this technique is that determines the application behavior during runtime and loads target data. The resource consumption in this analysis technique is more as compared to static analysis. Dynamic behavioral detection method constructs operation environment by using a sandbox, virtual machine, and other forms, and simulates the execution of the application to acquire the applications behavior model.

2.4.3 Hybrid Analysis

Hybrid Analysis is a combination of static and dynamic analysis. It is a technology or method that can integrate run-time data extracted from dynamic analysis into a static analysis algorithm to detect behavior or malicious functionality in the applications. The hybrid analysis method involves combining static features obtained while analyzing the application and dynamic features and information extracted while the application is executed. Though it could increase the accuracy of the detection rate, it makes the system cumbersome and the analysis process is time consuming.

2.5 State-of-the-art machine learning algorithms for Android malware detection

This section summarizes the most used machine learning classification algorithms in the literature related to Android malware detection and family classification.

2.5.1 Decision Trees

Decision trees are one of the former machine learning methods for regression and classification. They provide a useful mechanism based on a set of splitting rules. These models make a prediction based on the most common class in the region of the example. Typically, decision trees are generated through consecutive binary splitting while an error function is minimized. One of the strengths of these models lies in that they allow an easy interpretation and visualization. In contrast, they have several drawbacks, such as classification problems when presenting data with small changes. An example of decision tree algorithm is ID3, which makes divisions trying to maximize the information gain.

2.5.2 Random Forest

Decision trees are one of the simplest learning techniques. However, a decision tree tends to overfit the training data, since it is a literal interpretation of the data, and provides no generalization of the training set. To partially alleviate this problem, multiple decision trees can be used, where each is trained on a subset of the training data. A random forest takes this idea one step further by also training on subsets of the classifiers [13]. Although much of the inherent simplicity of decision trees is lost in this process, random forests have proved to be a very strong machine learning technique over a wide variety of applications.

2.5.3 K-nearest neighbors

The K-nearest neighbors is a non-parametric classifier, meaning that this model grows in parallel to the size of the training data [14]. Basically, it uses a distance metric to provide a prediction based on the majority class among the K nearest point to the example. While these models have proved to be powerful in varied problems, they are not indicated for high-dimensional spaces.

2.5.4 Support Vector Machine

This is a powerful algorithm used for both classification and regression problems. It is based on representing each example in a n-dimensional space, where a hyperplane is created pursuing the best separation between classes [15]. For building this hyperplane, a portion of the training data called support vectors is employed. These models have been widely used in the literature for building malware detection tools.

2.5.5 Logistic Regression

Logistic regression is the appropriate regression analysis to conduct when the dependent variable is dichotomous (binary). [16] Like all regression analyses, the logistic regression is a predictive analysis. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal,interval or ratio-level independent variables.

2.5.6 Gradient boosting

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

Nowadays we have some state-of-the-art algorithms implementing Gradient boosting. Below, we introduce the top 3 of the most used algorithm implementing Gradient boosting.

XGBoost

XGBoost is an algorithm that has recently been dominating applied machine learning and Kaggle competitions for structured or tabular data.

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance created by Tianqi Chen, now with contributions from many developers. [17]

LightGBM

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages: Faster training speed and higher efficiency, Lower memory usage, Better accuracy, Support of parallel and GPU learning, Capable of handling large-scale data. [18]

Catboost

CatBoost is an algorithm for gradient boosting on decision trees. It is developed by Yandex researchers and engineers, and is used for search, recommendation systems, personal assistant, self-driving cars, weather prediction and many other tasks at Yandex and in other companies, including CERN, Cloudflare, Careem taxi. It is in open-source and can be used by anyone. [19]

2.5.7 Artificial Neural Networks

Artificial neural networks are algorithms that can be used to perform nonlinear statistical modeling and provide a new alternative to logistic regression, the most commonly used method for developing predictive models for dichotomous outcomes in medicine. Neural networks offer a number of advantages, including requiring less formal statistical training, ability to implicitly detect complex nonlinear relationships between dependent and independent variables, ability to detect all possible interactions between predictor variables, and the availability of multiple training algorithms. [20]

Chapter 3

Methodology

In this chapter, we describe the methodology of our proposed Android malware detection system based on static features and how we implemented our models using our proposed Stacking method.

3.1 Dataset

To begin with this Chapter we will introduce the dataset used in this paper. We used a dataset called Androzoo. AndroZoo is a growing collection of Android Applications collected from several sources, including the official Google Play app market. It currently contains 10,299,119 different APKs, each of which has been (or will soon be) analyzed by tens of different AntiVirus products to know which applications are detected as Malware [21].

Due to keeping a balanced dataset containing both malware and benignware samples, We collected 10,000 malware APKs and 10,000 benignware samples randomly from the above dataset Androzoo. Also, we have done a filtering process in which we remove repeated applications and those that are considered as invalid (they cannot be actually installed and executed). After removing these applications we download again the remaining number to complete 10,000 of APKs and done the filtering process again. We repeated these steps until to gather a clean dataset consisted of 10,000 APKs of malware applications and 10,000 APKs of benignware applications with a total of 20,000 APKs.

3.2 Static Features extraction

Static features focus on a large part of the state-of-the-art literature related to Android malware detection. The easy and quick extraction of this kind of feature makes them suitable to be used to build malware detection tools.

One of our system goals is to do a prediction process while the user only needs to input the APK file to receive the output (malware or not). In other words, it is called an end-to-end system. One essential process to achieve this goal is, we have to automatically the feature extraction process and convert it to vectors that the machine learning models could use for inputs to predict something. So in this study, we use an opensource tool called AndroPytool [3].

AndroPyTool can extract static and dynamic features, this section focuses on the former ones. By using Android malware analysis tools such as AndroGuard or by analyzing the source code of each sample, AndroPyTool extracts a representative set of characteristics based onstate-of-the-art Android malware detection approaches. These features include API calls, mainactivity name, opcodes, package name, permissions, intent receivers, intents services, intent activities, strings found, system commands or information flows extracted with FlowDroid.

Once extracted the desired set of characteristics, such as permissions requested, API calls invoked or information flows, all this information is processed to build a representative feature vector for each application. This process is shown in Fig. 3.1 On the one hand, M measurable characteristics

$$C_M = \{c_1, c_2, \ldots, c_M\}$$

are assigned a value counting the number of occurrences of that feature within the application. On the other hand, K binary features

$$D_K = \{d_1, d_2, \ldots, d_K\}$$

point the use or the existence of a particular attribute. The bundling of both features composes the static description of the application behavior, in other words, our input dataset $\mathcal{D}$.

$$\mathcal{D} = \{\mathbf{x}_i, y_i\} \text{ where } \mathbf{x}_i = \left\{c^i_1, c^i_2, \ldots, c^i_M, d^i_l, d^i_2, \ldots, d^i_K\right\}$$

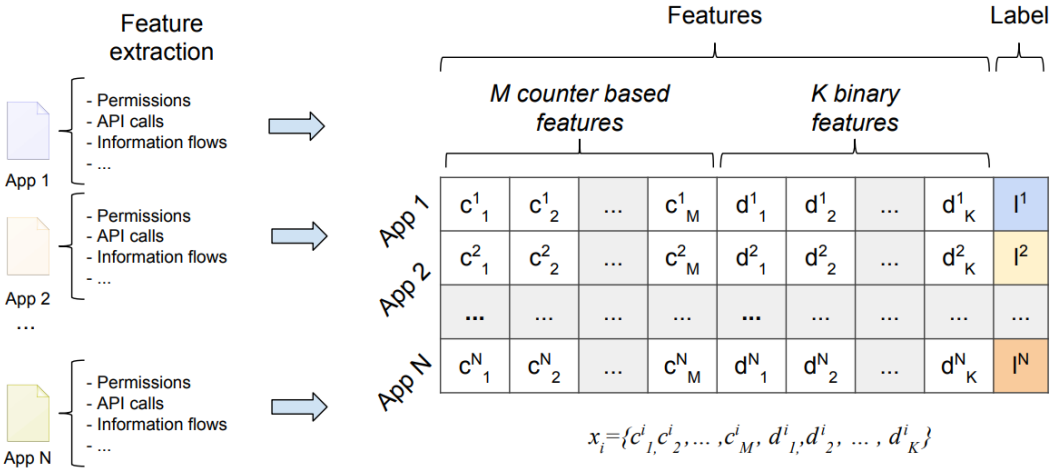

Figure 3.1: Extraction and representation of static information into representative feature vectors. [3]

3.3 Stacking

Stacking is an ensemble learning technique that combines multiple classifications or regression models via a meta-classifier or a meta-regressor [22]. The base level models are trained based on a complete training set, then the meta-model is trained on the outputs of the base level model like features as we can see in Figure 3.2.

The base level often consists of different learning algorithms and therefore stacking ensembles are often heterogeneous. But we must note that this type of Stacking is prone to overfitting due to information leakage. We should not derive the predictions for the 2nd-level classifier from the same dataset that was used for training the level-1 classifiers. The algorithm 1 summarizes stacking.

Besides, Stacking is a commonly used technique for winning the Kaggle data science competition. For example, the first place for the Otto Group Product Classification challenge was won by a stacking ensemble of over 30 models whose output was used as features for three meta-classifiers: XGBoost, Neural Network, and Adaboost.

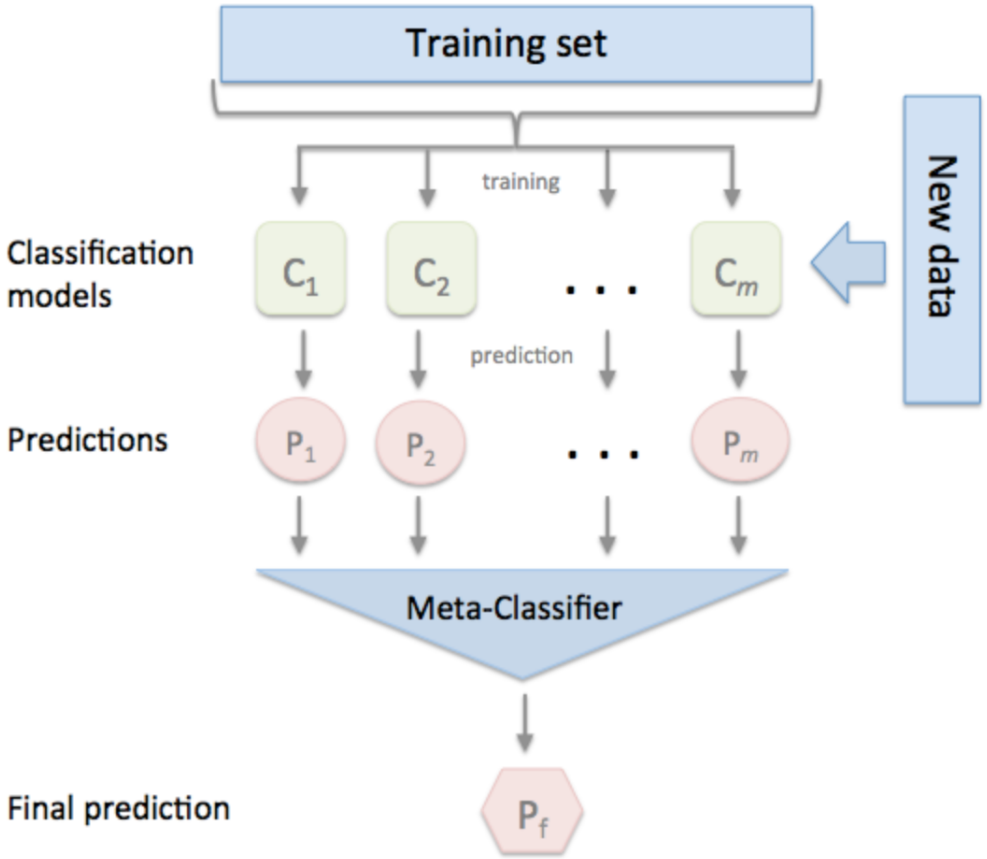

Figure 3.2: Stacking pipeline figure.

Algorithm 1 Stacking

Input: Training data $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^{m}$ $(\mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathcal{Y})$
Output: An ensemble classifier H

Step 1: Learn first-level classifiers
for $t \leftarrow 1$ to T **do**
  Learn a base classifier h_t based on $\mathcal{D}$
end for
Step 2: Construct new data sets from $\mathcal{D}$
for $i = 1$ to m **do**
  Construct a new data set $\mathcal{D}_h = \{\mathbf{x}_i', y_i\}$, where $\mathbf{x}_i' = \{h_1(\mathbf{x}_i), \ldots, h_T(\mathbf{x}_i)\}$
end for
Step 3: learn a meta-classifier
Learn a new classifier h' based on the newly constucted data set $\mathcal{D}_h$
return $H(\mathbf{x}) = h'(\mathcal{D}_h)$

3.4 Proposed model pipeline

In this section, we will introduce our proposed Stacking model pipeline as we can see in Figure3.3. Our proposed model consists of 3 stages with different models to predict one final output.

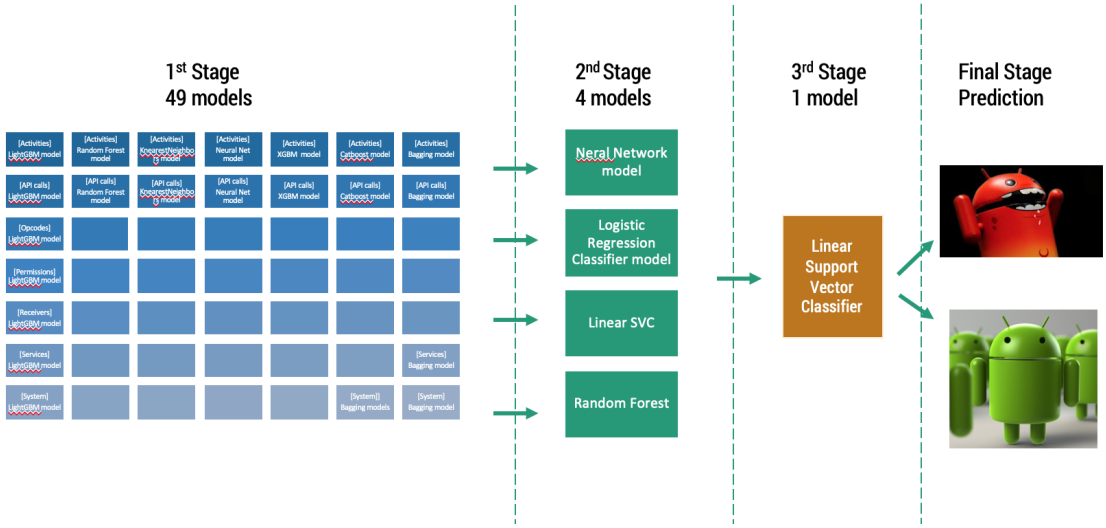

Figure 3.3: Proposed Stacking model pipeline figure.

3.4.1 1st Stage

For this first stage, we proposed to train 7 different algorithms (Bagging, XGBoost, K-neighbors, LogisiticRegression, RandomForest NeuralNetworks) to each 7 different static feature (Activities, API calls, Opcodes, Permissions, Receivers, Services, System commands) to create our models of a total of 49 models.

Following the Algorithm 1, we trained the base 49 classifiers $h_{t=49}$ on our training dataset as mentioned in section 3.1. Where our input training data is $\mathbf{x}_i$ and label data is y_i

$$\{\mathbf{x}_i, y_i\}_{i=1}^{m=7} \left(\mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathcal{Y}\right)$$

Due to every model has one output for one APK input, we gain one output vector from $\mathcal{D}$ as we can see at below

$$\mathcal{D}_h = \left\{\mathbf{x}_i', y_i\right\}, \text{ where } \mathbf{x}_i' = \left\{h_1\left(\mathbf{x}_i\right), \ldots, h_{T=49}\left(\mathbf{x}_i\right)\right\}$$

3.4.2 2nd Stage

In this stage, we proposed to train 4 different algorithms (Neural Network, Logistic Regression classifier, Linear SVC, RandomForest) with the output vector from the 1st stage as input data. Following the Algorithm 1, we train our 2nd stage meta-classifiers $H_k(k = 4)$ based on the newly constructed data set at 1st stage $\mathcal{D}_h$. Then we obtained the below models H_k

$$H_k(\mathbf{x}) = h'\left(\mathcal{D}_h\right)$$

3.4.3 3rd Stage

In this stage, we repeated the 2nd stage process using $H_k\left(\mathbf{x}\right)$ as input to train our final meta-classifier (Linear SVC).

We trained our final meta-classifier L based on the newly constructed dataset at 2nd stage $\mathcal{D}_l = \{\mathbf{h}'_i, y_i\}$. Then we obtain the below trained final meta-classifier $L(\mathbf{x})$

$$L(\mathbf{x}) = l'(H_1(\mathbf{x}), H_2(\mathbf{x}), \ldots, H_k(\mathbf{x}))$$

3.5 Proposed Malware detection System

In this section, we introduce our proposed malware detection system implementing our proposed Stacking model in section 3.4. We can see the overview of our proposed system pipeline in Figure 3.4.

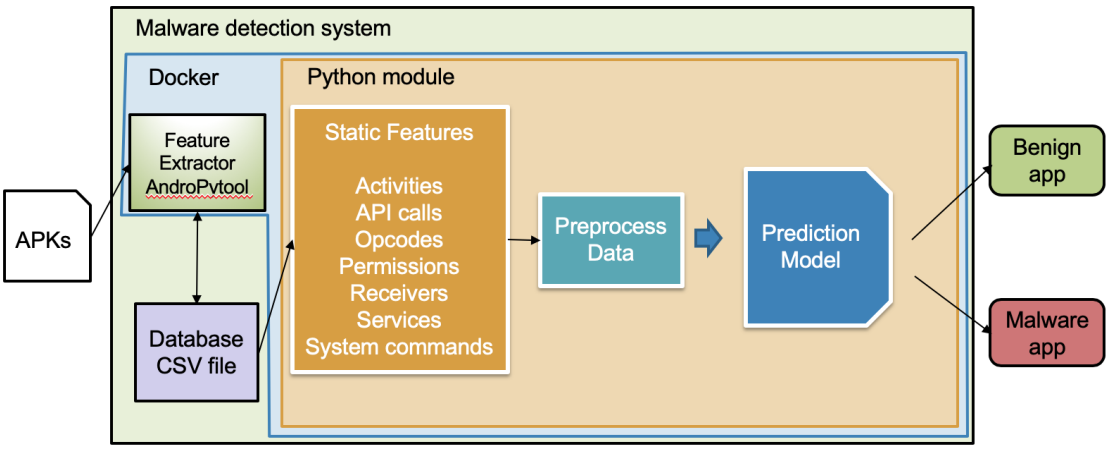

Figure 3.4: Proposed malware detection system pipeline figure.

3.5.1 AndroPytool module

We build almost all of this system in a Linux Docker container. The main merit of building a system with Docker is that it makes easy for the users to deploy this system in other environments not needing to care about dependent packages.

First, we implemented the AndroPyTool [3] module. Here we input the APK file to extract the static features to our database as Comma-separated values (CSV) file, as we can see in section 3.2.

3.5.2 Preprocessing data

Next, we load the extracted features from our database to our python module. In this module, we preprocess our data (static features in vectors) before to input it into our classifier model. We applied two steps to preprocess our data.

Standard scaling

In the first step, we applied the Standard scaling to all of our counter-based features. Below is the formula applied for Standard scaling.

$$x' = \frac{x - \bar{x}}{\sigma}$$

Where x is the original feature vector, $\bar{x} = average(x)$ is the mean of that feature vector, and σ is its standard deviation.

One-hot encoding

For the second step, we applied One hot encoding method to our categories features or binary features. One hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction. The figure 3.5 explain the One hot encoding process. In this example, we applied One hot encoding to the CompanyName collum. After converting the CompanyName to CategoricalValue it is converted to binary values.

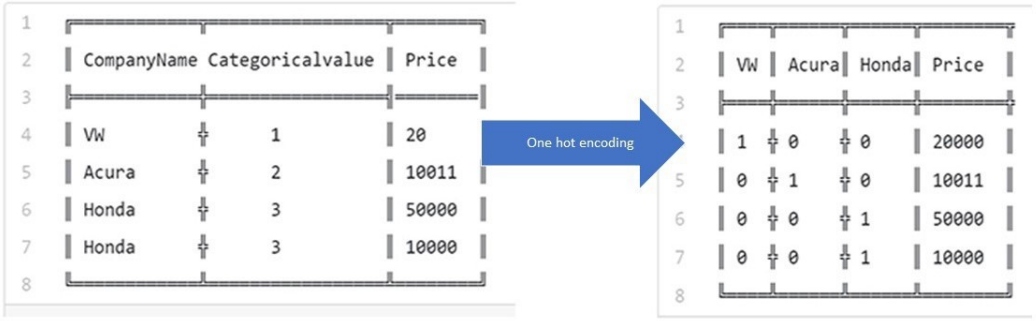

Figure 3.5: One hot encoding explained in an image.

3.5.3 Prediction of malware

After preprocessing our data to the best format for all of our classifiers, finally, we do the malware prediction step. We input the preprocessed data to the 49 models as mentioned in section 3.4. After the 3 steps of prediction, the final classifier outputs the final prediction (0 for Benign app, 1 for Malware app).

Chapter 4

Experiments and Results

The main aim of this section is to assess the use of static features when involved in the characterisation of Android malware and benignware samples used to train detection tools. Thus, through the use of several machine learning classifiers, it is studied if this combination with stacking methods is feasible and, if so, to assess its performance when applied to build a machine learning aided Android malware detection tools.

4.1 1st Stage training experiments

For the 1st stage training step of our model, the selected ensemble classifiers were run with the Scikit-learn library for Python [23]. They include AdaBoost, XGBoost, Catboost, Bagging Classifier, K-nearest neighbors, Neural network, Random Forest. All these ensemble classifiers were tested with the previously mentioned dataset through a 10-folds cross validation with accuracy scoring method showed below.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN)$$

$$where : TP = \text{True positive; } FP = \text{False positive;}$$

$$TN = \text{True negative; } FN = \text{False negative}$$

For a better evaluation of the capacity of these algorithms to extract and generalise conclusions, several experiments were conducted in order to judge the individual performance of the all models. The results are shown in Table 4.1 and Figure 4.1

Table 4.1: Experiments results on 1st training stage

Model	Activities CV	API calls CV	Opcodes CV	Permissions CV	Receivers CV	Services CV	System CV
LightGBM	0.5094	0.8783	0.8519	0.7979	0.8486	0.5094	0.7866
Bagging	0.5108	**0.8817**	0.8544	0.8094	0.8577	0.5099	0.7986
CatBoost	0.5103	0.8762	0.8499	0.8048	0.8525	0.5031	0.7912
Neural Network	0.5124	0.8518	0.8189	0.7841	0.8429	0.5092	0.7692
XGBoost	0.5099	0.8740	0.8469	0.8008	0.8388	0.5094	0.7809
Random Forest	0.5098	0.8238	0.8348	0.6143	0.7946	0.5064	0.7526
K-Neighbors	0.5050	0.8466	0.8279	0.7817	0.8440	0.5116	0.7743

In particular, API calls compose the most appropriate representation to build a machine learning classification tool, exhibiting a 88.17% accuracy with a Bagging classifier. In general, this algorithm obtains the best results overall.

It is remarkable the fact that not only API calls has good results but also Opcodes and Receivers have proved to be powerful at building detection mechanisms as we can see in Figure 4.1. In contrast, we found that Activities and Services are to be less useful at building malware detection mechanisms.

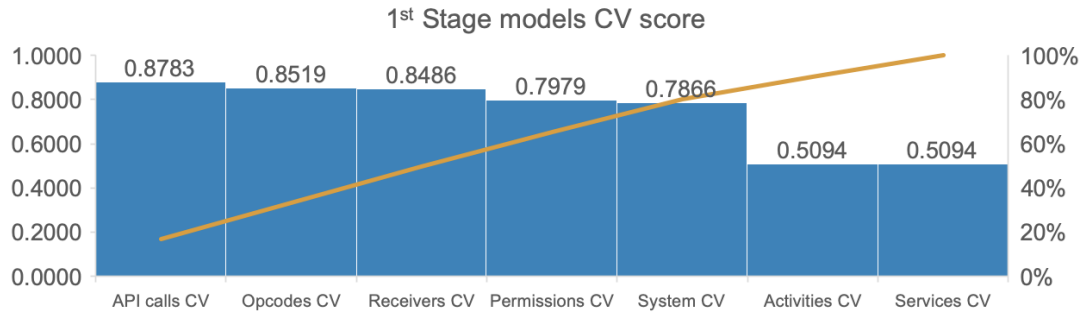

Figure 4.1: 1st Stage models CV accuracy score average

4.2 2nd Stage training experiments

In the 2nd stage for training we used the predictions results of the 1st stage as input to training our 4 selected classifiers. Due to we have only 49 inputs to each model, we selected 4 algorithms that are good to handle with small datasets. It are Neural Network, Logistic Regression Classifier, Linear SVC, Random Forest. All these classifiers were tested with the previously mentioned dataset through a 10-folds cross validation with accuracy scoring method as section 4.1. The results are shown in the Table 4.2 and Figure 4.2.

Table 4.2: Experiments results on 2nd training stage

Model	CV score
LinearSVC	**0.9356**
RandomForestClassifier	**0.9339**
Neural Network	**0.9334**
LogisticRegression	**0.9312**
1st Stage best model	0.8817

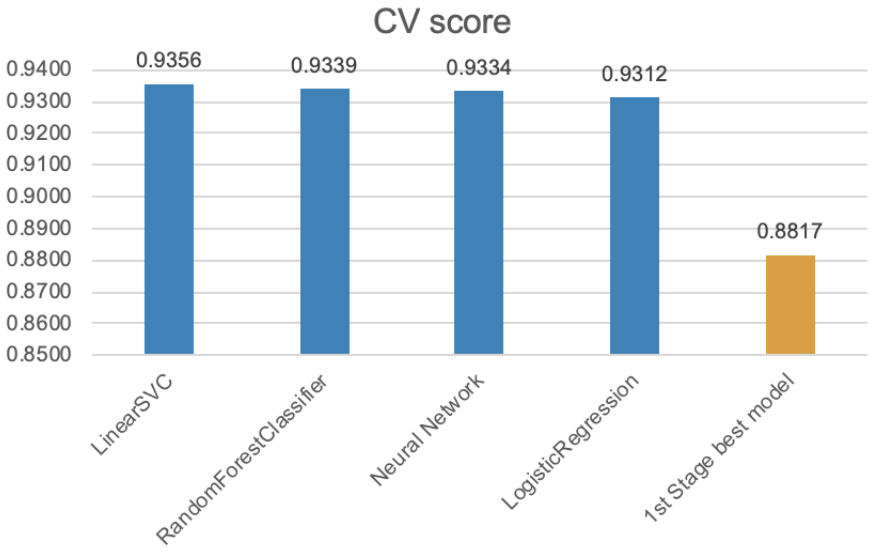

Figure 4.2: 2nd Stage models CV accuracy score

From these results of 2nd stage we could see that our all models score are better than the best score from 1st stage. Moreover, comparing to the best model at this stage we have a difference

of about 5.39% from the 1st stage best model. This fact could prove that our Stacking method is useful and very powerful method to improve the performance of malware detection systems or models.

References

[1] A. Holst. Market share of mobile operating systems worldwide 2012-2019. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[2] G. D. Raj Samani. Mobile malware continues to increase in complexity and scope. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf

[3] A. Martn Garca, R. Lara-Cabrera, and D. Camacho, "A new tool for static and dynamic android malware analysis," 09 2018, pp. 509–516.

[4] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dl-droid: Deep learning based android malware detection using real devices," *Computers and Security*, vol. 89, p. 101663, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404819300161

[5] Google. Google play. [Online]. Available: https://play.google.com/

[6] R. S. Gary Davis. Mcafee mobile threat report q1, 2018. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2018.pdf

[7] D. Arp, M. Spreitzenbarth, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," 2014.

[8] M. Yusof, M. M. Saudi, and F. Ridzuan, "A new mobile botnet classification based on permission and api calls," in *2017 Seventh International Conference on Emerging Security Technologies (EST)*, Sep. 2017, pp. 122–127.

[9] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: Detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 8, pp. 1772–1785, Aug 2017.

[10] Kaggle. [Online]. Available: www.kaggle.com

[11] Androguard. [Online]. Available: https://github.com/androguard/androguard

[12] Apktool. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[13] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002. [Online]. Available: https://CRAN.R-project.org/doc/Rnews/

[14] C. Robert, "Machine learning, a probabilistic perspective," *CHANCE*, vol. 27, no. 2, pp. 62–63, 2014. [Online]. Available: https://doi.org/10.1080/09332480.2014.914768

[15] T. Evgeniou and M. Pontil, "Support vector machines: Theory and applications," vol. 2049, 01 2001, pp. 249–257.

[16] What is logic regression ? [Online]. Available: https://www.statisticssolutions.com/what-is-logistic-regression/

[17] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *NIPS*, 2017.

[19] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: unbiased boosting with categorical features," 2017.

[20] J. V. Tu, "Advantages and disadvantages of using artificial neural networks versus logistic regression for predicting medical outcomes," *Journal of Clinical Epidemiology*, vol. 49, no. 11, pp. 1225 – 1231, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0895435696000029

[21] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[22] "Stacking classifier." [Online]. Available: http://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/#overview

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Mller, J. Nothman, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and douard Duchesnay, "Scikit-learn: Machine learning in python," 2012.