Efficient Classification of Android Malware in the wild using Robust Static Features

Hossein Fereidooni[§], Veelasha Moonsamy[†], Mauro Conti[§], Lejla Batina[†]
[§]University of Padua, Italy
[†]Radboud University, The Netherlands
{hossein, conti} @math.unipd.it
{veelasha, lejla} @cs.ru.nl

1 Abstract

The ubiquitous use of Android smartphones continue to threaten the security and privacy of users' personal information. Its fast adoption rate makes the smartphone an interesting target for malware authors to deploy new attacks and infect millions of devices. Moreover, the growing number and diversity of malicious applications render conventional defenses ineffective. Thus, there is a need to not only better understand the characteristics of malware families but also, to generate features that are robust and efficient for classification over an extended period of time.

In this chapter, we propose a machine learning based malware detection and classification methodology, with the use of static analysis as feature extraction method. Our tool, **uniPDroid** can be used to extract a plethora of informative features from our extensive dataset. We performed a malware family classification and obtained an average classification accuracy of 92%. We also present the empirical results for our cumulative classification which investigates how well features from old malware can contribute to the detection of new variants of both known and unknown malware.

2 Introduction

Since its first release in late 2008[1], Android smartphones have continuously been replacing the traditional mobile phones. The advent of such high-powered and affordable smart devices has redefined the way mobile phone users carry out their day-to-day activities. From checking emails to doing online banking, mundane tasks once conducted on a desktop only are now being executed "on the go". According to Gartner[2], worldwide sale of Android smartphones in 2015 has reached more than 271 million devices, which accounted for 82.2% of the market share. Due to its popularity, the number of malware targeting the Android platform has increased significantly in recent years. As such, malicious applications pose a significant threat to the smartphone platform security. In the first half of 2014, F-Secure[3] reported that 295 new threat families or new variants of known families were collected. It is also worth mentioning that 294 out of these 295 families

[1]http://www.cnet.com/news/a-brief-history-of-android-phones/
[2]http://www.gartner.com/newsroom/id/3115517
[3]https://www.f-secure.com/documents/996508/1030743/Threat_Report_H1_2014.pdf

run on Android platform. Additionally, in the first quarter of 2015, Kaspersky's mobile security products detected $103,072$ new malicious applications, a three-fold increase from last quarter of 2014 [15].

On one hand, these statistics further prove that Android continues to be a favorite target for majority of the mobile threats, as smartphones continue to replace traditional phones. On the other hand, the security of Android platform still requires thorough understanding, as demonstrated by the plethora of attacks in [11, 13, 14, 26]. Thus, effective ways of enforcing security on such devices are still subject to investigation and there exists further room for improvement. To address the aforementioned security issue, we can leverage various techniques to analyze and detect Android malicious applications.

The techniques used to detect Android malware are similar to the ones used on other platforms. Detection techniques are essentially broken into: (i) static analysis by analyzing a compiled file, (ii) dynamic analysis by analyzing the runtime behavior, and (iii) hybrid analysis by combining static and dynamic techniques [19]. *Static analysis* refers to extraction and analysis of information about an application from binary, source code or other associated files. Static analysis can be performed before executing the application for the first time. However, this method is rendered ineffective by obfuscation techniques as it is not able to deal with malware sample that changes its code without changing functionality, such as polymorphic malware.

On the other hand, *dynamic analysis* relies on execution of code in a virtual environment or sandbox to monitor the interaction of applications with the operating system. This approach comes with several drawbacks: (i) it is not clear how long the monitoring period should be in order to detect key important events, (ii) it is not always evident which conditions trigger the malicious behavior and (iii) dynamic analysis might be more resource-consuming and computationally expensive than static analysis.

In the early days, malware detection and classification mechanisms employed only either static or dynamic analysis for feature extraction and malware prediction. However, as malicious programs continued to evolve in complexity and to deploy sophisticated attacks, there was a need for more robust frameworks. Thus, applying a hybrid method, which is a combination of static and dynamic analysis as shown in [25], when building the feature vector space is considered as one way of dealing with this problem. It should be noted that selecting a hybrid method when dealing with smartphone malware is not a popular method as this technique requires high computational resources and could impact negatively on the desired seamless interaction between the user and the device.

In a nutshell, static analysis is beneficial on memory-limited Android-powered devices because the malware is not executed and only analyzed. Additionally, static analysis makes use of reverse engineering tools to extract information from an application. For these reasons, we will concentrate on the lightweight approach and thus, advocate for static analysis through the use of machine learning.

Machine Learning (ML) techniques to detect mobile malware have been extensively investigated, leveraging a few characteristics of the mobile applications (for example, call graphs [16], permissions [6], or both API calls and permissions [2, 22]), and the results obtained were promising. Classification approaches have also been proposed to model and approximate the behaviors of Android applications and discern malicious apps from benign ones. The detection accuracy of a classification method depends on the quality of the features (for example, how specific the features are [10]). Grace et al. [18] proposed a classification method with pure static features (data and control-flow analysis) that gives a False Negative (FN) rate of 9%. Zhou et al. [39] extracted hybrid features, that is a combination of static and dynamic features, obtaining a better FN rate of 4.2%.

Although it is critical to distinguish malicious applications from clean ones, it is also important to efficiently classify malware into their correct families. Malware authors often redistribute repackaged version of existing malware and therefore, by correctly classifying the original malware, it becomes easier for antivirus engines to detect repackaged versions. Moreover, the features used to classify malware should also be robust and relevant over a long period of time as out-of-date features would allow malware samples to evade

detection and classification mechanisms. To address the aforementioned issues, we focus solely on malicious applications to *firstly* investigate how to efficiently and accurately classify malware samples into their correct families, and *secondly* generate robust feature sets that will stand the test of time and still be relevant over a period of years; this is tested through the experimental work referred to as cumulative classification.

In this chapter, we propose a malware classification method to: (i) leverage an extensive coverage of applications' behavioral characteristics than the state-of-the-art; (ii) integrate decision-making through multiple classifiers; and (iii) utilize the robustness of extracted features to detect and classify newly-discovered malware. Specifically, we utilize a large number of features, extracted statically, from our extensive dataset comprising of $15,884$ samples. We extract Intents, actual permissions used by an application, critical API calls, Linux system commands, and some other features that could possibly indicate the presence of malicious behaviors in an application. In order to build our classifier, we utilize the *eXtreme Gradient boost* (XGboost[4]) classifier, which is an ensemble method where weaker learners are combined to make a stronger learner. XGboost contains a modified version of the Gradient Boosting algorithm and can automatically do parallel computation with OpenMP, and it is much faster than the existing Gradient Boosting algorithm. Our aim is to maximize the accuracy scores of our classifier in terms of F1-score, Recall and Precision.

In particular, our main contributions can be summarized as follows:

- We presented an Android malware detection and classification method that uses several informative features with good discriminative power to categorize malicious apps under their respective family names. We designed and built a tool named **uniPDroid**, written in Python programming language to extract the features such as Intents, permissions used by an app, critical API calls, Linux system commands, and some other features that might indicate capability of performing malicious activities by an app.
- We performed an extensive static analysis on large-scale well-labelled dataset of $15,884$ Android applications. The dataset includes malware developed within a seven-year period, from year 2009 to 2015 and collected from different well-known and reliable repositories.
- We used several ML classification algorithms to discover the most highly performing one in terms of accuracy and speed. We leveraged boosting techniques to obtain as much detection and classification performance as possible for Android malware detection in the wild. Our experimental evaluations show that our proposed detection method is very effective and efficient. It obtained a true positive rate in detecting malware applications as high as **92%**.

This chapter is organized as follows: in Section 3, we present the related work in the area of malware detection and classification. Section 4 provides an extensive description of the proposed classification framework, including the dataset collection and pre-processing, feature extraction and selection, and evaluation metrics used. In the next section, we then present the experimental work for malware family-based classification followed by the work on cumulative classification in Section 6. In Section 7, we provide our conclusions.

[4]https://github.com/dmlc/xgboost

3 Literature Review

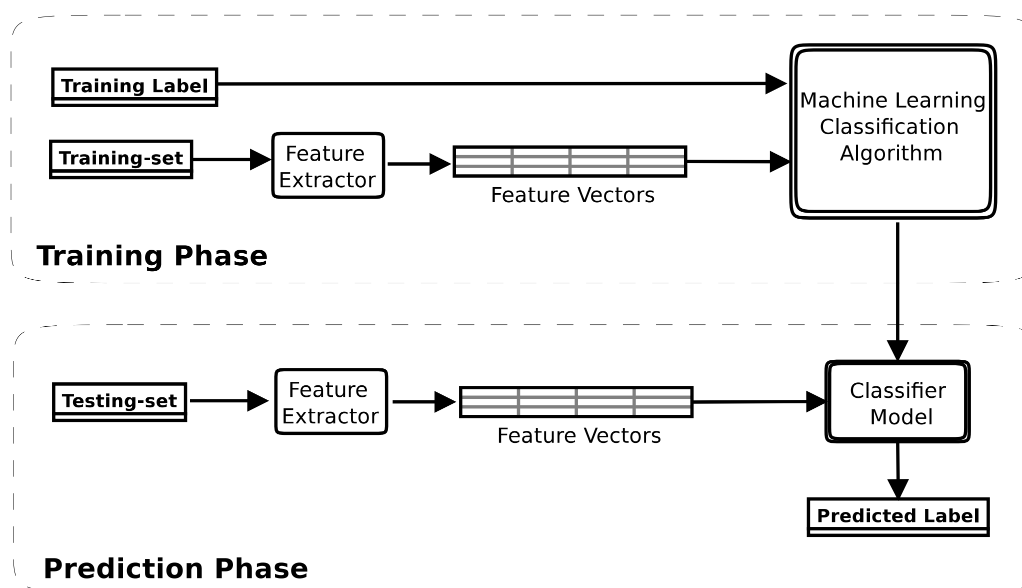

Figure 1: General classification methodology

Machine Learning (ML) techniques have been extensively used for detection of malware on mobile devices [6, 24, 36]. Figure 1 presents an overview of the general framework of a standard ML technique - classification. In the remainder of this section, we present some of the existing work in the area of Android malware classification.

The authors in [28] applied clustering techniques in malware detection of Android applications. They extracted the features of the applications from the application's XML file, which contains permissions requested by apps then applied unsupervised ML techniques to detect malware applications automatically. Similarly, Arp et al. [5] presented Drebin, an on-device malware detection tool utilizing ML-based methods on features such as requested hardware components, permissions, names of application components, intents, and API calls. Gascon et al. [16] presented a method that disassembles applications and extracted their function call graphs using the Androguard framework. They also proposed learning-based method for the detection of malicious Android applications. Their method employed an explicit feature map inspired by the neighborhood hash graph kernel to represent applications based on their function call graphs.

Allix et al. [3] have used several ML classifiers to build a set of features in the form of Control Flow Graphs (CFG) of applications to classify benign from malicious applications. The authors focused exclusively on the history aspect of datasets used in their experiment rather than malware detection performance. Karim et al. [10] proposed a classification approach to detect malware by extracting a data dependence graph representing inter-procedural flows of data. The authors extracted a data-flow feature on how user inputs can trigger sensitive API invocations.

The authors of [9] suggested a solution to detect Android malware collusions by constructing Inter-Component Communication (ICC). The authors constructed ICC maps to capture pairwise communicating ICC channels of $2,644$ Android applications. Britton et al. [35] extracted the frequencies of all possible

n-byte sequences in the Android application's bytecode as features and trained several classification algorithms to classify benign applications from malicious ones. The authors used a dataset comprising of $3,869$ Android applications. Sahs et al. [27] presented an ML-based framework for Android malware detection using Support Vector Machines (SVM) algorithms. The authors exploited a single-class SVM model derived from benign samples. They used the Android permissions in the Manifest files and CFGs of applications from the dataset. `Crowdroid` [7] collects behavioral-related data directly from users via crowd-sourcing and evaluates the data with a clustering algorithm.

Shabtai et al. [33] proposed a new method for categorizing Android applications through ML techniques. To represent each application, their method extracts different feature sets including the frequency of occurrence of the printable strings, the different permissions of the application itself, and the permissions of the application extracted from the Android Market. Abela et al. [1] presented `AMDA`, an automated malware detection system for the Android platform. The authors extracted features such as system calls form benign and malware applications to provide baseline behavior datasets to feed machine learners. Test applications are then passed through the behavior-based module for identification of presence of malicious payloads. Similarly, `RobotDroid` [37] is a framework that detects smartphone malware based on SVM active learning algorithm. The authors in [30] designed an anomaly detection system that extracts the strings contained in application files in order to detect malware. Their proposed method is based on features that were extracted from string analysis of the application.

Martinelli et al. [12] proposed `CAMAS`, a framework for the analysis and classification of malicious Android applications, through pattern recognition on execution graphs. They extracted a subset of frequent subgraphs of system calls that are executed by most of the malware. The resulting vector of the subgraphs is given to a classifier that returns its decision in terms of whether or not a malware has been detected. `DroidAnalytics` [38] is a malware analytic system for malware collection, signature generation and association based on similarity scores by analyzing the low-level system at the application, class or method level.

The authors in [31] proposed another detection method for Android malware. In particular, they used only manifest files to detect malware. The proposed method extracts six types of information from manifest files such as Permission, Intent (action, priority and category), Process name and Number of redefined permission and then uses them to detect Android malware. `DroidMat` presented by Wu et al. [36], exploits permissions, intents, inter-component communication, and API calls to distinguish malicious apps from benign ones. The detection performance was evaluated on a dataset of $1,500$ benign and 238 malicious applications and compared with the `Androguard` risk ranking tool, with respect to detection metrics such as accuracy rate.

The work in [23] presented a machine learning approach including SVM, Decision Trees (DT), and Bagging predictor to detect malicious Android applications. They trained and tested a classifier by using extracted permissions and API calls as features to identify whether an application is potentially malicious or not. Koundel et al. [20] designed a Naive Bayes classifier to classify applications using various attributes of an application, such as the permissions used by an application, battery usage and rating acquired by the application on Android market. `MAMA` [29] presents Manifest analysis for malware detection in Android. It extracts several features from the Android Manifest of the applications to build machine-learning classifiers such as K-Nearest Neighbors, DT, SVM and Bayesian networks.

The literature presented in this section provides an overview of the existing work in the field of Android malware versus cleanware detection and ML-based classification methodologies. In our work, we focused solely on Android malware, proposing a novel ML-based methodology that can efficiently, and with high accuracy, assign malware samples to their correct family names. We argue that it is not only important to detect malicious applications, but also to label them correctly as malware authors often repackage existing

malware. Hence, re-detecting these repackaged samples becomes easier if the correct family names are used. Additionally, we analyzed the robustness of our extracted features used by the proposed methodology by performing a cumulative malware classification. We verified how efficient are features extracted from old malware samples in terms of detecting and classifying newly discovered malware.

4 Proposed Classification Framework

This section provides extensive details on how the experimental dataset was collected and pre-processed, feature extraction and selection, and a description of the classification models and evaluation metrics used for the empirical results. In Section 3.1, we describe the composition of our experimental dataset, followed by an explanation of the different types of features extracted in Section 3.2. In Section 3.3, we elaborate on the methodology used for selecting the most representative features used by our classification model - Section 3.4. Finally, in the last subsection, we provide more details on the evaluation metrics used for our empirical results.

4.1 Dataset Collection and Pre-processing

In this subsection, we provide further detail on the composition of our experimental dataset. In order to conduct an extensive analysis, we collected a set of large well-labeled Android malicious applications. The dataset used in our evaluation is composed of $15,884$ malicious applications collected from the following existing work in the literature: [5,21,34,40]. The samples were released over a period of seven years, starting from 2009 until 2015. Table 1 shows the details of the dataset composition.

Repository	Number of samples
Genome [40]	1,260
Drebin [5]	5,560
M0Droid [21]	193
VirusTotal [34]	8,871
Total	**15,884**

Table 1: Dataset composition

To perform malware classification using supervised machine learning classification algorithm (for example, XGBoost classifier), we are required to provide a well-labelled dataset. To find the class label associated with each malware sample in our dataset, we wrote several scripts in `Bash` and `Python` programming languages. We submitted each malware sample to *Virustotal* [34] and made a query to get the malware family name, as shown in listings 1 and 2. Virustotal then returned an analysis report for the given file in the form of `JSON` object as depicted in Listing 3. We then parsed the `JSON` object and performed text processing to extract the related family names. The names were then used as class labels since there is no agreed-upon malware naming convention among antivirus (AV) companies.

In order to decide on the family name for each class label, we took into account the family names of top eight AV engines[5]. Leveraging these top eight AVs and based on majority voting role, we extracted the selected malware family names. The AVs that we exploited are among the top AV engines used on the Android platform and are namely: *MicroWorld-eScan, BitDefender, Kaspersky, Avira, AVG, Emsisoft,*

[5] http://www.av-comparatives.org/wp-content/uploads/2014/03/security_survey2014_en.pdf

AVware, and *F-Secure*. The reason for considering only these eight AVs is because (i) they are among top AV engines dedicated to the Android Platform; (ii) we observed that these AV outperform others in most cases, particularly when detecting malware; and (iii) we did not further complicate the text processing phase by increasing the number of AV engines.

```python
#imports
import simplejson
import urllib
import urllib2

url = "https://www.virustotal.com/vtapi/v2/file/report"
parameters = {"resource":APK-hasH-name,"apikey":apikey}
data = urllib.urlencode(parameters)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
json-object = response.read()
print json-object
```

Listing 1: Example of a Python script for submitting malware samples to Virustotal

```
{"scans": {
"Kaspersky":{"detected":true,"version":"15.10","result":"Trojan-Spy.AndroidOS.Adrd.a",..},
"BitDefender":{"detected":true,"version":"7.2","result":"Android.Trojan.Adrd.A",..},
"Emsisoft":{"detected":true,"version":"3.5.0.642","result":"Android.Trojan.Adrd.A",.. },
"F-Secure":{"detected":true,"version":"11.0.19100.45","result":"Trojan:Android/Adrd.A",..},
"Avira":{"detected":true,"version":"8.3.2.4","result":"ANDROID/Spy.Adrd.D.Gen",..},
.
.
"AVG":{"detected":true,"version":"16.0.0.4489","result":"Android/Adr",..},
"resource": "4de0d8997949265a4b5647bb9f9d42926bd88191", "total": 54, "positives": 38,
"md5": "77b0105632e309b48e66f7cdb4678e02",...}
```

Listing 2: Example of a JSON file produced by Virustotal

4.2 Feature Extraction

Android applications are written in Java, compiled to Java bytecode, and then converted into platform-specific Dalvik bytecode. This bytecode can be efficiently disassembled and provides us with useful information about features used in an application. We mainly extracted the features from bytecode and converted these features into binary feature vectors, which are made up of 560 features. Each feature vector is comprised of the features described below:

- *Intents*: the intent is an abstract description of an operation to be performed and allowing information about events to be shared among different components and applications. We extract all intents in Android app as a feature set because malware often listen to specific intents. Listing 3 shows the snippet used to extract intents from an application.

```
from androguard.core.bytecodes.dvm import *
from androguard.core.bytecodes.apk import *
from androguard.core.analysis.analysis import *

a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
for i in range(len(z)):
  if z[i].startswith('android.intent.action.'):
    intents = z[i]
    intentList.append(intents)
```

Listing 3: Example of a Python script for extracting intents from Dalvik bytecode

- *Used permissions*: a significant part of Android's built-in security is its permissions system. Permissions allow an application to access potentially dangerous API calls. Many applications need several permissions to function properly and user must accept them at install-time. The used permission provides a more in-depth view on the behavioral characteristics of an application. We extract and include them to the feature set (e.g., INTERNET, ACCESS_FINE_LOCATION, INSTALL_PACKAGES). Listing 4 describes the permissions extraction process.

```
...
# the APK
a = APK("app.apk")
# the classes.dex
d = dvm.DalvikVMFormat( a.get_dex() )
# the analyzed classes.dex
dx = analysis.uVMAnalysis( d )

Permission_dexFile = dx.get_permissions( [] )
for i in Permission_dexFile:
  permList.append(i)
```

Listing 4: Example of a Python script for extracting permissions from Dalvik bytecode

- *System Commands*: malware use system commands to run root exploit code or download and install additional executable files. Since system command can provide us with valuable information to detect malicious behavior, we extract and include them in the feature set. The authors in [32] listed the most commonly used system commands in malicious applications (for example, `chmod, su, mount, sh, killall, reboot, mkdir, ln, ps`). These commands are executed after the malware gains root privilege on the device. Listing 5 shows how the system commands are extracted from Dalvik bytecode.

```
...
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
# to back trace unix commands
suspicious_cmds=["su","mount","reboot","mkdir"
                 ,...]
for i in range(len(z)):
  for j in range(len(suspicious_cmds)):
    if suspicious_cmds[j]==z[i]:
      cmdList.append(suspicious_cmds[j])
```

Listing 5: Example of a Python script for extracting system commands from Dalvik bytecode

- *Suspicious API calls*: we extracted the API calls that are frequently seen in malware samples and can result in malicious behavior. In order to obtain a deeper understanding of the functionality of an application, we collected these API calls and included them in the feature set (for example, `openFileOutput`, `sendTextMessage`, `getPackageManager`, `getDeviceId`, `Runtime.exec`, `Cipher.getInstance`). The authors in [32] mentioned the most commonly used API calls in malicious applications. Listing 6 shows the snippet of code used to extract suspicious API calls.

```
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
z = d.get_strings()
suspicious_APIs=["getSimSerialNumber",
      "getSubscriberId","getDeviceId",...]
for i in range(len(z)):
  for j in range(len(suspicious_APIs)):
    if suspicious_APIs[j]==z[i]:
      APIsList.append(suspicious_APIs[j])
```

Listing 6: Example of a Python script for extracting suspicious API calls form Dalvik bytecode

- *Malicious Activities*: we considered different malicious behaviors seen in malware applications. We investigate whether an Android application is capable of performing such malicious activities through Dalvik bytecode analysis. We consider different kinds of information that malicious applications are able to harvest from smartphones. In Listing 7, we describe how to search for features that are prone to perform malicious activities. We briefly list some of these features below.
  - Reading the IMEI
  - Loading Native, Dynamic, and Reflection Code
  - Accessing files on SD card
  - Reading location information through GPS/WiFi
  - Intercepting data network activities
  - Making phone calls and disabling incoming SMS notifications
  - Retrieving information of the application installed
  - Recording audio and capturing video
  - Opening a TCP/UDP Socket
  - Performing encryption and message digest algorithms

```
...
# Searching for Doing Cipher
a = APK("app.apk")
d = dvm.DalvikVMFormat( a.get_dex() )
dx = analysis.uVMAnalysis( d )
getIN=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","getInstance",".")
ScrKey=dx.tainted_packages.search_methods
("Ljavax/crypto/spec/SecretKeySpec","<init>",".")
Cipherini=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","<init>", ".")
CipherDO=dx.tainted_packages.search_methods
("Ljavax/crypto/Cipher","doFinal", ".")
if ((getIN)and(Cipherini)and(CipherDO))or(ScrKey):
  potential_misBhve.append('Does Cipher')
```

Listing 7: Example of a Python script for extracting potential misbehaviour from Dalvik bytecode

4.3 Feature Selection

We should consider that a large number of features, some of which are redundant or irrelevant, may present several problems such as misleading the learning algorithm, over-fitting, and increasing model complexity. Feature selection is a process which automatically selects features in dataset that contribute most to the prediction results. The benefits of performing feature selection before modelling the data are to reduce over-fitting, to improve accuracy, and to reduce training time. We used a technique leveraging ensemble of randomized decision trees, that is, Extra-Trees Classifier for determining the feature importances [17]. We exploited Extra-Trees Classifier to compute the relative importance of each attribute to help better the feature selection process. We used a meta-transformer, SelectFromModel [17], for selecting features based on importance weights as shown in Listing 8. This feature transformer can be used along with any estimator that has a `feature_importances_` attribute after fitting. If the corresponding features' importance values are below the user-defined threshold parameter (for example, Mean), the features are considered as unimportant and consequently, are discarded. Figure 2 shows the most important features (total of 101 binary features) that we used to train and evaluate our classification algorithms.

```
#To build a forest
clf = ExtraTreesClassifier(n_estimators=600)
clf = clf.fit(X_train, y_train)
#To compute the feature importances
importances=clf.feature_importances_
# To reduce 560 features to 101
model = SelectFromModel(clf, prefit=True)
X_train_new = model.transform(X_train)
```

Listing 8: Example of a Python script used for feature selection process

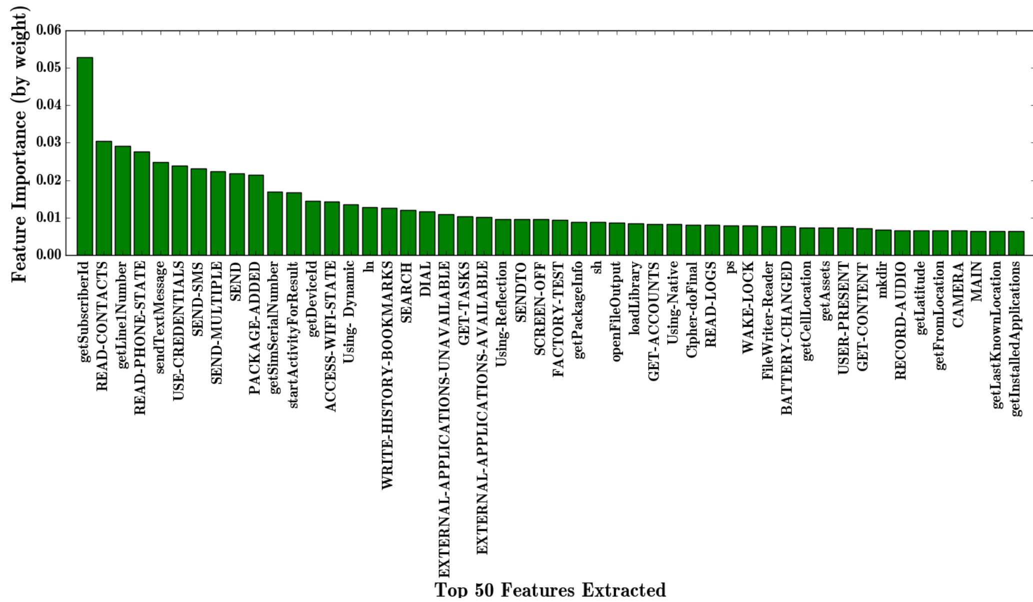

Figure 2: Key features extracted from our dataset

4.4 Classification Models

XGBoost [8] is the abbreviation for eXtreme Gradient Boosting. It is a gradient boosting tree method. *Gradient* refers to the use of gradient descent, which can be used as a way to find a local minimum of a function and *Boosting* is a technique which consists of the fact that a set of weak learners is stronger than a single strong learner. XGboost algorithm uses a differentiable loss function to calculate the adjustments needed to be made to a consecutive successor learner in an iterative learning sequence. The algorithm can automatically do parallel computations with OpenMP (an API for writing Multi-threaded Applications), and it is much faster than existing Gradient Boosting algorithm. Listing 9 provides an excerpt of the source code for XGBoost.

```python
import numpy as np
import xgboost as xgb
from sklearn.metrics import  classification_report

def train():
  data_train = np.genfromtxt(open("train.csv","r"), delimiter=",")
  y_train = data_train[:,0]
  X_train = data_train[:,1:]
  xg_train = xgb.DMatrix(X_train, label=y_train)
  data_test = np.genfromtxt(open("test.csv","r"), delimiter=",")
  y_test = data_test[:,0]
  X_test = data_test[:,1:]
  xg_test = xgb.DMatrix(X_test, label=y_test)
  # setup parameters for xgboost
  param = {}
  param['objective'] = 'multi:softmax'
  param['eta'] = 0.1
  param['max_depth'] = 6
  param['silent'] = 1
  param['nthread'] = 4
  param['num_class'] = 78 # Number of classes starting from 0
  watchlist = [ (xg_train,'train'), (xg_test, 'test') ]
  num_round =  260
  bst = xgb.train(param, xg_train, num_round, watchlist);
  # get prediction
  y_pred = bst.predict( xg_test );
  print classification_report(y_test, y_pred)

if __name__ == '__main__':
  train()
```

Listing 9: Example of code for the Machine Learning classifier, eXtreme Gradient Boosting

The different parts of the proposed classification methodology, explained in previous subsections, can be summarized in Figure 3. We extended the `Androguard` tool [4] and built `uniPDroid`, a static analysis tool written in `Python` programming language. Our proposed method uses this tool to extract several informative features representing characteristics of the application and leverages several `Python` ML libraries to build the best performing classifier, XGBoost, in order to perform classification task. In particular, the system consists of two modules: (i) Feature Extraction Module, and (ii) Machine Learning Classification Module. The feature extraction module includes three components. The `uniPDroid.py` is the main component within this module extracting informative features from an application while `Androguard` and `Androlyze.py` are auxiliary components providing support for performing feature extraction task. The ML classification module leverages several ML packages to perform classification. The main component within this module is the `MalClassifier.py`. The `Scikit-learn` and `REP` packages provide different classification algorithms and some helper functions for performance evaluation.

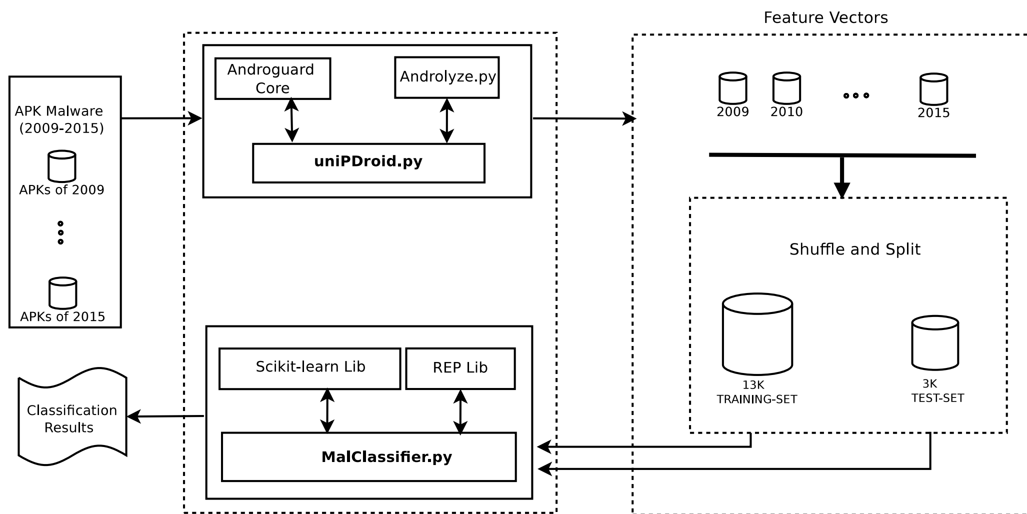

Figure 3: Framework of proposed classification methodology

4.5 Evaluation metrics

Table 2 introduces the metrics that we considered in order to assess the performance of the ML classification algorithms in class imbalance problem, that is, the total number of a class of positive data is far less than the total number of negative data. The highest precision means that an algorithm returns substantially more relevant results than irrelevant ones, while the highest recall means an algorithm returns the most of the relevant results. The F1-score combines precision and recall: it is the harmonic mean of precision and recall. We elaborate further on our empirical results in Section 6.

Metric	Description	Formula
Precision	Measure of exactness or quality	$\frac{T_P}{T_P+F_P}$
Recall	Measure of completeness or quantity	$\frac{T_P}{T_P+F_N}$
F1-score	Harmonic mean of precision and recall	$2\times\frac{Precision\times Recall}{Precision+Recall}$

Table 2: Performance metrics

5 Malware Family-based Classification

In this experiment, we carried out family by family malware classification. To this end, we grouped $15,884$ Android malware in our repository into 204 different malware families. To perform an efficient and effective classification task and have sufficient samples to feed our proposed ML classification algorithm, we discard malware families that include less than 10 samples and consequently, ended up with 78 malware families. We shuffled and split the whole data points into training and testing sets, 80% and 20% respectively. We levereged XGBoost classification algorithm to perform classification task over the 78 different malware families. Tables 3, 4, and 5 show the malware families used in our experiments as well as the infection risks associated with each malware family.

Before conducting classification task, in order to achieve a high accuracy in performance, we fine-tuned the hyper-parameters of our classification algorithm (e.g., learning rate, the number of decision trees and their maximum depth) through Grid-Search procedure combined with 5-fold Cross-Validation over the training set. Having the best parameters selected, we trained our classifier on the training set (using $13,000$ samples) and tested its performance against $3,000$ unseen samples in classifier point of views (that is, on the testing set). Table 6 shows the classification results (F1-score) for each malware family and Table 8 illustrates the overall accuracy measures in terms of Precision, Recall, and F1-score over the 78 malware families.

Family Name	Infection Risks
AdFlex	An advertisement library may compromise your personal information
ADRD	Steals private information
Adwo	An advertisement library may compromise your personal information
Agilebinary	A Spyware accessing the file system and retrieving app data
AirPush	A very aggressive Ad-Network and compromises your personal information
Andup	Steals personal information
AppQuanta	An advertisement library may compromise your personal information
Asroot	Uses Asroot root exploit
AutoSMS	Attempts to steal sensitive data by seizing incoming SMS messages and forwards them to a remote site
BaseBridge	Sends premium-rate SMS to predetermined numbers
Boxer	Sends SMS to premium-rated numbers
Cobbler	A monitoring tool and wipes the SD card's contents and everything stored on the device
DDLight	Collects information about the device and sends back to a remote server
Dianjin	An advertisement library which may compromise your personal information
Dianle	Interrupts the normal operations and gains access to private information
Dougalek	Steals personal information and uploads these data to a remote server
Downloader	Gains root access and downloads additional malicious apps
DroidSheep	Captures and hijacks unencrypted web sessions
Dropper	Interrupts the normal operations and gains access to private information
Ewalls	Steals information from the mobile device
Exploid	Exploits vulnerabilities to gain root privileges on devices
FakeApp	Downloads configuration files to display advertisements and collects information from the compromised device
FakeBank	Opens a back door and steals information from the compromised device
FakeDoc	Installs additional applications
FakeInstall	Pretends to be an installer for legitimate app, sends premium-rate SMS
FakeTimer	Sends personal information to a remote server and opens pornographic websites

Table 3: Infection risks associated with each malware family

Family Name	Infection Risks
Feejar	Sends SMS to premium-rated numbers
Geinimi	Opens a back door and transmits private information
Gepew	Attempts to replace installed apps with trojanized versions
GingerBreak	A root exploit for Android 2.2 and 2.3
GingerMaster	Utilizes a Root Exploit and provides root-level access
GoldDream	Steals information from Android devices
GoneSixty	Steals private information
Hamob	An advertisement library may compromise your personal information
HiddenAds	Does not have an icon and runs in a stealth mode and displays various advertising messages
Igexin	An advertisement library may compromise your personal information
InfoStealer	Secretly collects and uploads sensitive information
JSmsHider	Opens a backdoor and sends information to a specific URL
Kmin	Attempts to send data to a remote server
Kuguo	An advertisement library may compromise your personal information
KungFu	Forwards confidential information to a remote server
LeadBolt	An advertisement library may compromise your personal information
Lovetrap	Sends SMS to premium-rated numbers and steals information
Mecor	Monitors and compromises your personal information
Metasploit	Exploits vulnerabilities to gain root privileges on devices
Minimob	Compromise personal information and distributes via spam email
Mobclick	Aggressively pushes unwanted ads and steals personal information
MobileTX	Steals information from the compromised device and may send SMS to a premium-rate number
Mseg	Steals private data and secretly send SMS to premium-rated numbers
MTK	Interrupts the normal operations and gains access to the private information
Mulad	Generates income by injecting ads into legitimate free apps
NickiSpy	Gathers information from infected user's smartphone and uploads the data to a specific URL

Table 4: Infection risks associated with each malware family (continued)

Family Name	Infection Risks
NoiconAds	Compromises personal information
Pentr	A Spyware and hack-tool enables penetration testing
RuFraud	Sends SMS to premium rated numbers
SecApk	An advertisement library that compromises your personal information
SLocker	Encrypts images, documents and videos in the SD Card to later ask for a ransom to decrypt the files
SMSKey	Interrupts the normal operations and gains access to the private information
SmsPay	Mimics a legitimate app and requires an activation fee through SMS
SMSReg	Registers the infected user to non-free services
SMSSend	Reaps profit by silently sending SMS to premium-rate numbers
SmsSpy	Attempts to steal sensitive data by seizing incoming SMS and forwards them to a remote site
SMSZombie	Exploits a vulnerability in the mobile payment system used by China Mobile
SndApps	Compromises your personal information
SpyHasb	Monitors phone calls, SMS, and GPS locations
SpyPhone	Steals personal data
Steek	A fraudulent app advertising an online income solution and steals privacy related information and sends SMS
Tekwon	Interrupts the normal operations and gains access to the private information
Utchi	An advertisement library may compromise your personal information
Vdloader	Steals personal information
Viser	Opens back door by use of the system loopholes to introduce some adware, browser extensions, spyware or ransomware
Wallap	Promises access to a wide collection of wallpapers and uses ads libraries to generate revenue
Waps	An advertisement library may compromise your personal information
Wapz	An advertisement library may compromise your personal information
Youmi	An advertisement library may compromise your personal information
YZHCSMS	Sends SMS to a premium-rate number
Zdtad	An advertisement library may compromise your personal information
Zsone	Sends SMS to premium rated numbers

Table 5: Infection risks associated with each malware family (continued)

Table 6 shows the results of classification per malware family, number of samples, the year that those samples have been developed, and the percentage of malware families represented in our dataset. According to the table, the malware families such as `SMSReg`, `FakeInstall`, `SMSPay`, `Kungfu`, and `Mulad` have the biggest share of malware samples in the entire dataset, 12.3%, 10.8%, 8.4%, 6.6%, and 6.3% respectively. The worst classification results, 40%, belongs to `Minimob` family with 14 samples. It is obvious that by increasing the number of samples in the training set, our proposed ML classification algorithm will be expected to perform the training procedure better. It can be noted in Table 6, as the size of the training set for each malware family increases (that is, number of samples in each family), the accuracy (F1-score) gets better. In other words, with a few amount of samples it is not reasonable to expect to achieve good prediction accuracies from the classification algorithm.

Family Name	Samples	Year Developed	Percentage of apps	Classification F1-score (%)
AdFlex	68	2013	0.40	86
ADRD	59	2010	0.37	100
Adwo	388	2011	2.4	83
Agilebinary	10	2010	0.06	100
AirPush	787	2010	4.9	93
Andup	18	2013	0.11	100
AppQuanta	39	2013	0.24	100
Asroot	12	2009	0.07	100
AutoSMS	46	2013	0.28	75
BaseBridge	608	2010	3.8	97
Boxer	21	2010	0.13	77
Cobbler	15	2011	0.09	100
DDLight	124	2011	0.78	100
Dianjin	91	2012	0.57	91
Dianle	54	2012	0.33	77
Dougalek	22	2012	0.13	93
Downloader	75	2012	0.47	83
DroidSheep	11	2011	0.06	100
Dropper	123	2014	0.77	95
Ewalls	43	2009	0.27	100

Family Name	Samples	Year Developed	Percentage of apps	Classification F1-score (%)
Exploit	41	2010	0.25	100
FakeApp	104	2011	0.65	80
FakeBank	84	2014	0.52	96
FakeDoc	130	2011	0.81	100
FakeInstall	1729	2011	10.8	98
FakeTimer	21	2012	0.13	100
Feejar	12	2014	0.07	50
Geinimi	152	2010	0.95	100
Gepew	13	2014	0.08	100
GingerBreak	14	2011	0.08	67
GingerMaster	489	2011	3	90
GoldDream	126	2011	0.8	77
GoneSixty	15	2011	0.09	100
Hamob	35	2012	0.22	80
HiddenAds	44	2014	0.28	91
Igexin	42	2011	0.26	91
InfoStealer	209	2010	1.3	91
JSmsHider	11	2009	0.07	100
Kmin	187	2010	1.1	99
Kuguo	84	2012	0.52	50

Family Name	Samples	Year Developed	Percentage of apps	Classification F1-score (%)
KungFu	1051	2011	6.6	98
LeadBolt	178	2011	1.1	77
Lovetrap	11	2010	0.07	100
Mecor	10	2015	0.06	100
Metasploit	23	2014	0.14	100
Minimob	14	2013	0.09	40
Mobclick	101	2010	0.63	71
MobileTX	69	2011	0.43	100
Mseg	20	2011	0.12	67
MTK	97	2013	0.61	100
Mulad	1008	2012	6.3	99
NickiSpy	11	2010	0.07	100
NoiconAds	882	2014	5.5	99
Pentr	13	2011	0.08	67
RuFraud	21	2011	0.13	93
SecApk	59	2012	0.37	50
SLocker	22	2014	0.13	100
SMSKey	34	2011	0.21	100
SmsPay	1331	2010	8.4	88
SMSReg	1916	2010	12.3	88

Family Name	Samples	Year Developed	Percentage of apps	Classification F1-score (%)
SMSSend	487	2010	3	84
SMSSpy	207	2010	1.3	89
SMSZombie	18	2012	0.11	100
SndApps	23	2011	0.14	100
SpyHasb	13	2010	0.08	100
SpyPhone	23	2010	0.14	91
Steek	28	2011	0.17	91
Tekwon	16	2013	0.10	86
Utchi	26	2012	0.16	100
Vdloader	17	2012	0.10	77
Viser	36	2012	0.22	100
Wallap	88	2012	0.55	92
Waps	570	2011	3.5	78
Wapz	231	2012	1.5	75
Youmi	588	2010	3.7	82
YZHCSMS	59	2010	0.37	100
Zdtad	396	2015	2.5	99
Zsone	31	2011	0.19	86

Table 6: The number of malware samples, year developed and classification results of 78 malware families from our experimental dataset

Additionally, the average accuracy in terms of precision, recall, and F1-score for all 78 malware families are reported in Table 7. We conducted a 10-fold Cross-Validation experiment to compute Mean Error Rate for both training and testing sets. Table 8 shows the results obtained from this experiment. For the 10-fold cross-validation, the data is randomly partitioned into 10 equal size subsamples. Of the 10 subsamples, a single subsample is retained as the validation data for testing the model ($Test_{cv}$), and the remaining 9 subsamples are used as training data ($Train_{cv}$). The process is then repeated 10 times, with each of the 10 subsamples used exactly once as the validation data. The 10 results from the folds can then be averaged to produce a single estimation.

	Precision	Recall	F1-score	Support
Avg / Total	92	92	92	3000

Table 7: Classification Report (%) for test set (unseen samples)

	$Train_{cv}$ Mean Error Rate	$Test_{cv}$ Mean Error Rate
CV 10-Fold	0.033359 (+/- 0.000760)	0.091460 (+/- 0.007298)

Table 8: Cross-Validation result for training set

Comparing the F1-score, as shown in Table 7, which has been obtained from evaluating our proposed classifier against unseen samples with $Test_{cv}$, Mean Error rate and the prediction from Cross-Validation (which is equal to 92% accuracy), we can draw this conclusion that our ML classification algorithm is never over-fitted and is able to predict unseen samples with high accuracy rate.

6 Cumulative Classification

In this experiment, we accumulated Android malware apps and carried out cumulative classification where the classification results are continuously updated as new malware samples are discovered. The number of malware used in our experiment is $15,884$ samples. Figure 4 depicts the number of malware collected by month within the period 2009 and 2015 and Figure 5 shows the cumulative graph of the malware apps collected each month for that same period. In our cumulative classification, we used 56 different malware groups.

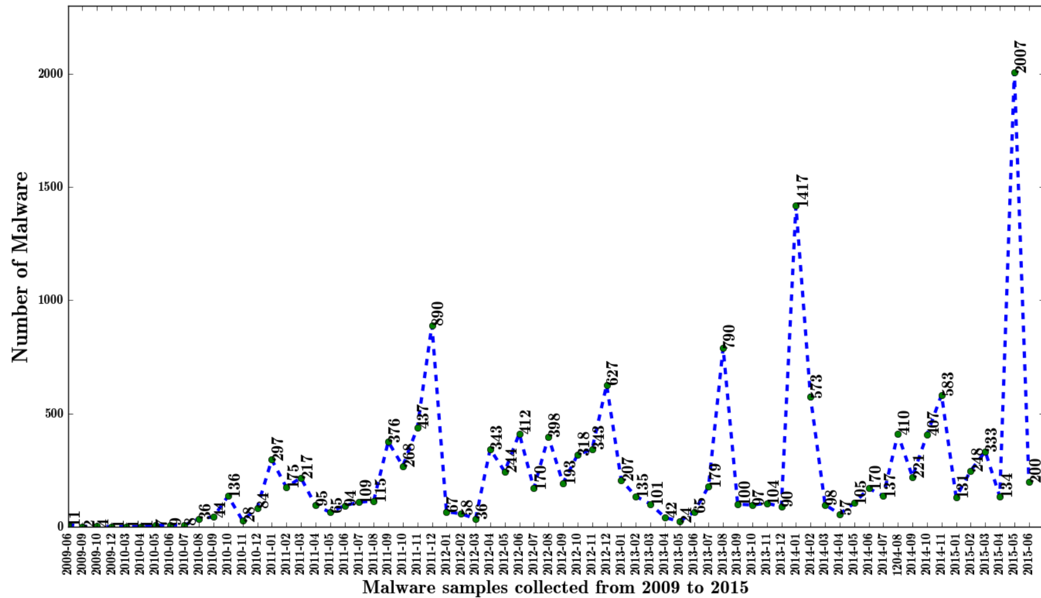

Figure 4: Malware number per month

To generate the first malware group, MG_1, we take the malware apps from June 2009 and September 2010 which comprises of 124 samples in order to have an initial set of samples enough to perform classification. The second data group, MG_2, contains the malware from June 2009 up to October 2010; this is achieved by adding malware belonging to upcoming month to previous months to generate the next malware group). For MG_3, we take the malware from June 2009 up to November 2010. The process is repeated until all the malware in the dataset are incorporated into the malware groups. Finally, we ended up with having 56 groups, MG_1,..., MG_{56} altogether, as shown in Figure 5.

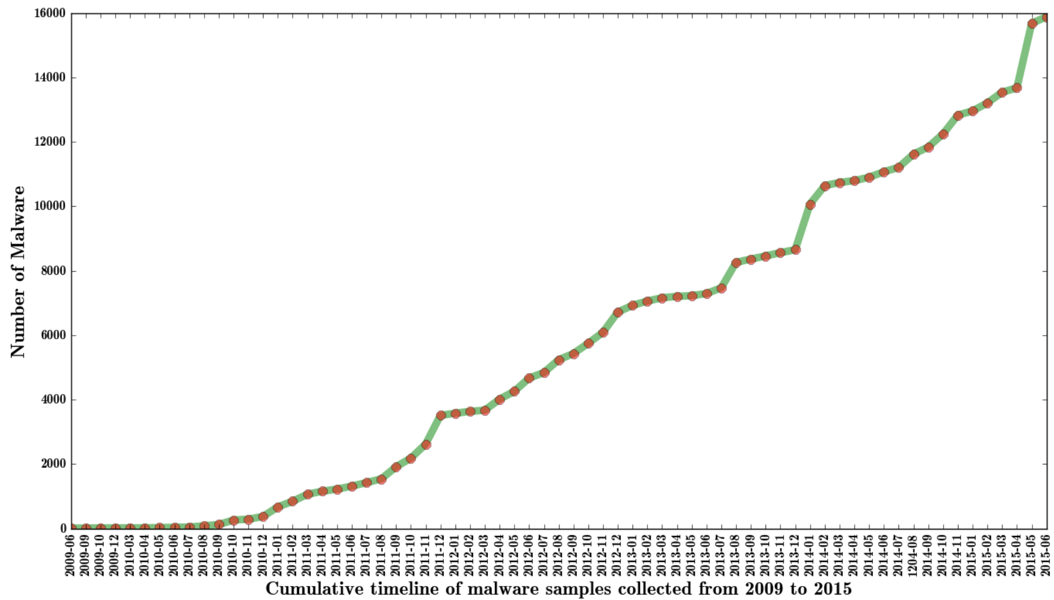

Figure 5: Malware number per month

We trained the classification algorithm, XGBoost, on each malware group, MG_i and tested its performance against malware belonging to upcoming months. We should take this point into account that malware belonging to the next month is unseen for the classification algorithm. We computed accuracy measures in terms of Precision, Recall, and F1-score, Figure 6. The aim was to investigate how features of old malware samples can be of help to classify new variant of both known and unknown malware families.

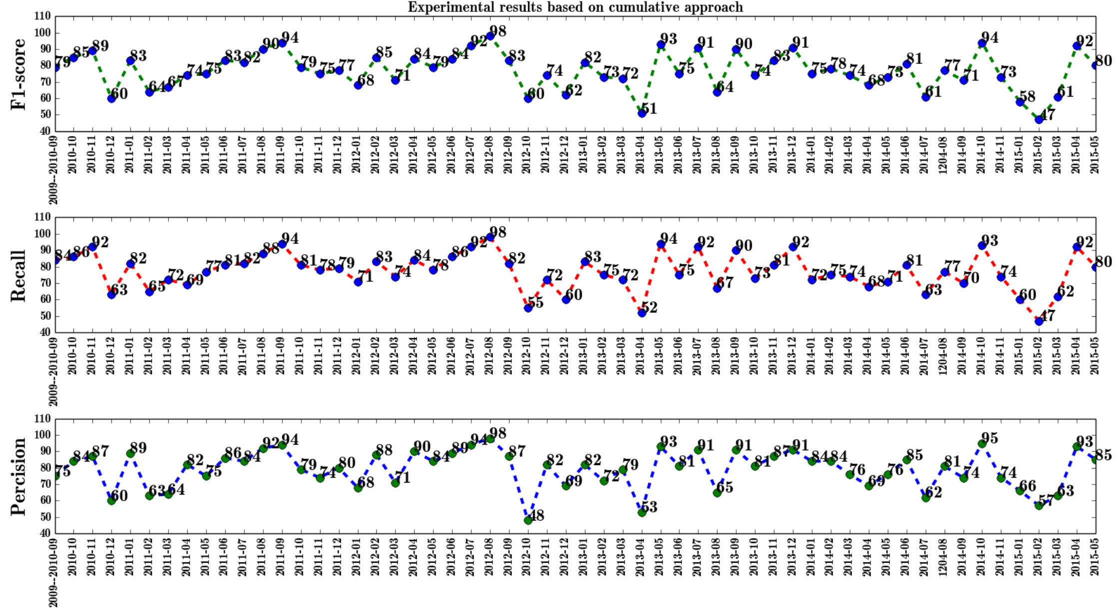

Figure 6: Accuracy measures

We performed cumulative classification to investigate how well the old malware can help us to detect new malware. In the other word, how old malware can contribute to detecting new variant of both known and unknown malware families. As for the accuracy measures obtained from cumulative classification, Figure 6, at some points (e.g., February 2015) accuracy measures drops. The reason for such a decrease in classifier performance is that we have trained the ML algorithm in certain time on data-points belonging to past up to that time and we evaluate its performance against future data-points. In the testing dataset, there exist some samples which are considered as zero-day malware in the wild (that is, recently developed malware). The ML classification algorithm has not been trained on such samples and has no idea about these malware samples which have completely different patterns in terms of features. Consequently, the classifier cannot predict the correct label of these samples based on its past experience. As it can be seen, in next round of cumulative classification by adding the old samples and enriching the training set we let the classifier learn more about past data and as a result the classifier might perform better during classification stage.

7 Conclusion

In this chapter, we proposed an ML-based malware detection and classification methodology together with the application of static analysis on an extensive dataset of Android applications. To this end, we designed a tool, **uniPDroid**, to extract as many informative features as possible from our dataset. We considered mainly features from the Dalvik bytecode. The features extracted were converted into feature vectors, each containing 560 binary features. We then applied feature selection on the aforementioned extracted features, which led to the selection of 101 informative binary features suitable to feed our proposed classification methodology.

Moreover, we performed an extensive Grid-search analysis along with a 10-fold Cross-validation to tune

the hyper-parameters of the classification algorithm to maximize the prediction accuracy. We performed Family-by-Family classification and obtained an average accuracy score of 92% in classification of unseen malware. In addition to this, we conducted a cumulative classification in order to investigate how well old malware can contribute to the detection of new variants of both known and unknown (zero-day) malware. We achieved reasonable accuracy rate, hence proving the robustness of the features extracted.

As future work, we will extend our Android application analysis and combine static analysis with features extracted from dynamic analysis to compensate the limitation associated with static analysis and get the best of both static and dynamic analyses. We will extract more features related to the behavior of Android applications such as CPU and Memory consumption, Network traffic activities, Inter-Process Communications (IPC) and system calls made by applications so as to interact with Android OS. In addition to these, we will make use of classification algorithms such as SVM along with different kernels to deal with structured data (e.g., string, set, graph, etc.).

8 Acknowledgments

Lejla Batina and Veelasha Moonsamy are supported by the Technology Foundation STW (project 13499 - TYPHOON & ASPASIA), from the Dutch government.
Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), the Italian MIUR-PRIN TENACE Project (agreement 20103P34XC), and by the projects "Tackling Mobile Malware with Innovative Machine Learning Techniques", "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua.

References

[1] K.J. Abela, D. K. Angeles, J. R. D. Alas, R. J. Tolentino, and M. A. Gomez. An Automated Malware Detection System For Android Using Behavior-based Analysis - AMDA. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(2):1–11, 2013.

[2] M. Alazab, V. Moonsamy, L. Batten, P. Lantz, and R. Tian. Analysis of malicious and benign Android applications. In *Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW 2012)*, pages 608–616, Macau, China, June 2012.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Are your training datasets yet relevant? *Engineering Secure Software and Systems*, 8978:51–67, 2015.

[4] Androguard. `https://code.google.com/p/androguard/`.

[5] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS 2014)*, pages 1–15, California, USA, February 2014.

[6] Z. Aung and W. Zaw. Permission-based android malware detection. *International Journal of Scientific and Technology Research*, 2(3):228–234, 2013.

[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Mobile Devices (SPSM 2011)*, pages 15–26, Illinois, USA, October 2011.

[8] Tianqi Chen. `https://github.com/dmlc/xgboost`.

[9] Karim Elish, Danfeng Yao, and Barbara Ryder. On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions. In *Proceedings of Mobile Security Technologies (MoST 2015)*, pages 1–5, California, USA, May 2015.

[10] K.O. Elish, X. Shu, D. Yao, B. Ryder, and X. Jiang. Profiling User-Trigger Dependence for Android Malware Detection. *Computers & Security*, 49:255–273, March 2015.

[11] Eurograbber. `https://www.checkpoint.com/download/downloads/products/whitepapers/Eurograbber_White_Paper.pdfv`.

[12] A. Saracino F. Martinelli and D. Sgandurra. Classifying android malware through subgraph mining. In *Proceedings of 6th International Workshop on Autonomous and Spontaneous Security (SETOP 2013)*, pages 1–15, Egham, UK, September 2013.

[13] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M.S. Gaur, M. Conti, and M. Rajarajan. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[14] E. Fernandes, B. Crispo, and M. Conti. FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment. *IEEE Transactions on Information Forensics and Security*, 8(6):1027–1037, 2013.

[15] M. Garnaeva, V. Chebyshev, D. Makrushin, and A. Ivanov. IT Threat Evolution in Q1 2015. `https://securelist.com/analysis/quarterly-malware-reports/69872/it-threat-evolution-in-q1-2015/`.

[16] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and Security (AISec 2013)*, pages 45–54, 2013.

[17] GitHub. Scikit-learn. `https://github.com/scikit-learn/`.

[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys 2012)*, pages 281–294, Lake District, UK, June 2012.

[19] N. Idika and A. P. Mathur. A survey of malware detection techniques. *Purdue University*, 2007.

[20] D. Koundel, S. Ithape, V. Khobaragade, and R. Jain. Malware classification using naives bayes classifier for android os. *The International Journal of Engineering And Science (IJES)*, 3:59–63, 2014.

[21] M0droid. `http://m0droid.netai.net/modroid/`.

[22] N. Peiravian and X. Zhu. Machine learning for android malware detection using permission and API calls. In *Proceedings of the 25th International Conference on Tools with Artificial Intelligence (ICTAI 2013)*, 2013.

[23] N. Peiravian and X. Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI 2013)*, pages 300–305, USA, November 2013.

[24] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec 2013)*, pages 1–6, Prague, Czech Republic, April 2013.

[25] B. P. S. Rocha, M. Conti, S. Etalle, and B. Crispo. Hybrid Static-Runtime Information Flow and Declassification Enforcement. *IEEE Transactions on Information Forensics & Security*, 8(8):1294–1305, 2013.

[26] G. Russello, M. Conti, B. Crispo, and E. Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT 2012)*, pages 3–12, Newark, USA, June 2012.

[27] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Proceedings of the 2012 European Intelligence and Security Informatics Conference (EISIC 2012)*, pages 141 –147, Odense, Denmark, August 2012.

[28] A.A. Samra, O.A. Ghanem, and K. Yim. Analysis of clustering technique in android malware detection. In *Proceedings of the 7th International Conference on Innovative Mobile and Internet Services on Ubiquitous Computing (IMIS 2013)*, pages 729–733, Taichung, Taiwan, July 2013.

[29] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P.G. Bringas, and G. Álvarez. Mama: Manifest analysis for malware detection in android. *Cybernetics and Systems, Intelligent Network Security and Survivability*, 44:469–488, 2013.

[30] B. Sanz, I. Santos, X. Ugarte-Pedrero, C. Laorden, J. Nieves, and P. G. Bringas. Anomaly detection using string analysis for android malware detection. In *Proceedings of the 6th International Conference on Computational Intelligence in Security for Information Systems (CICIS 2013)*, pages 1–10, Salamanca, Spain, September 2013.

[31] R. Sato, D. Chiba, and S. Goto. Detecting android malware by analyzing manifest files. *Proceedings of the Asia-Pacific Advanced Network*, 36:23–31, 2013.

[32] S-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim. Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications*, 38:43–53, 2014.

[33] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[34] VirusTotal. `http://www.virustotal.com`.

[35] B. Wolfe, K. Elish, and D. Yao. High precision screening for android malware with dimensionality reduction. In *Proceedings of the 13th International Conference on Machine Learning and Applications (ICMLA 2014)*, pages 21–28, Detroit, USA, December 2014.

[36] D.J. Wu, C.H. Mao, T.E. Wei, H.M. Lee, and Wu K. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Proceedings of the 7th Asia Joint Conference on Information Security (Asia JCIS 2012)*, pages 62–69, Tokyo, Japan, August 2012.

[37] M. Zhao, T. Zhang, F. Ge, and Z. Yuan. RobotDroid: A Lightweight Malware Detection Framework on SmartPhones. *Journal of Networks*, 7(4):1–8, 2012.

[38] M. Zheng, M. Sun, and J.C.S. Lui. DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2013)*, pages 163–171, Melbourne, Australia, July 2013.

[39] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Network andDistributed System Security Symposium (NDSS)*, pages 1–13, San Diego, USA, February 2012.

[40] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. *2012 IEEE Symposium on Security and Privacy (SP)*, pages 95–109, 20-23 May, 2012.