UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

BACHELOR THESIS

SIGNAL PROCESSING FOR MALWARE ANALYSIS

Computer Engineering Department

AUTHOR: Raquel Tabuyo Benito

TUTOR: Pedro Peris Lopez

June, 2016

"Perseverance is not a long race.
It is many short races one after the other"
-Walter Elliot

Acknowledgements

To my whole family, specially my sister, for whom I have an unconditionally love. I am really grateful for their dedication, patience, support and encouragement to follow my dreams.

To Pedro, my Bachelor Thesis tutor, whose kindness and guidance have helped me during this wonderful trip.

To my friends, thank you very much for showing me the meaning of true friendship.

Without all of you, this would have never been possible.

Abstract

This Project is an experimental analysis of Android malware through images. The analysis is based on classifying the malware into families or differentiating between goodware and malware. This analysis has been done considering two approaches. These two approaches have a common starting point, which is the transformation of Android applications into PNG images. After this conversion, the first approach was subtracting each image from the testing set with the images of the training set, in order to establish which unknown malware belongs to a specific family or to distinguish between goodware and malware. Although the accuracy was higher than the one defined in the requirements, this approach was a time consuming task, so we consider another approach to reduce the time and get the same or better accuracy. The second approach was extracting features from all the images and then using a machine learning classifier to get a precise differentiation. After this second approach, the resulting time for 100,000 samples was less than 4 hours and the accuracy 83.04%, which fulfill the requirements specified.

To perform the analysis, we have used two heterogeneous datasets. The Malgenome dataset which contains 49 kinds of malware Android applications (49 malware families). It was used to perform the measurements and the different tests. The M0droid dataset, which contains goodware and malware Android applications. It was used to corroborate the previous analysis.

Resumen

Este proyecto es un análisis experimental de aplicaciones de Android mediante imágenes. Este análisis se basa en clasificar las imágenes en familias o en diferenciarlas entre goodware o malware. Para ello, se han considerado dos enfoques. Estas dos aproximaciones tienen como punto en común la transformación de las aplicaciones de Android en imágenes de tipo PNG. Después de este proceso de transformación a imágenes, la primera aproximación se basó en restar cada imagen perteneciente al grupo de pruebas con las imágenes del grupo de entrenamiento, de esta forma se pudo saber la familia a la que pertenecía cada malware desconocido o distinguir entre aplicaciones goodware y malware. Sin embargo, a pesar de que la precisión de acierto era más alta que la definida en los requisitos, este enfoque era una tarea que consumía mucho tiempo, así que consideramos otra aproximación para reducir el tiempo y conseguir una precisión parecida o mejor que la anterior. Este segundo enfoque fue extraer las características de las imágenes para después usar un clasificador y así obtener una diferenciación precisa. Con esta segunda aproximación, conseguimos un tiempo total menor a las 4 horas para 100000 muestras con una precisión del 83.04%, cumpliendo y superando de esta forma los requisitos que habían sido especificados.

Este análisis se ha llevado a cabo usando dos sets de datos heterogéneos. Uno de ellos fue el perteneciente a un proyecto llamado Malgenome, éste contiene 49 tipos de familias de malware en Android. El set de datos de Malgenome se usó para realizar los diferentes ensayos o pruebas y sobre el que se realizaron las medidas de tiempo y precisión. El set de datos de M0droid se usó para corroborar el análisis previo y así establecer una clasificación final.

Table of Contents

1. Introduction **17**
1.1. Purpose 17
1.2. Motivation 17
1.3. Scope 19
1.4. Structure of the report 20
2. State of art **21**
2.1. Malware Evolution and Classification 21
2.1.1. Malware Evolution 21
2.1.2. Malware Classification and Android examples 21
2.2. Security Mechanisms 23
2.2.1. Security Mechanisms Overview 23
2.2.2. Security Mechanisms in Android 24
2.3. Malware Detection and Analysis 25
2.3.1. Detection Strategies 25
2.3.2. Contemporary Detection Systems 27
2.3.3. Detection System Android Examples 28
2.4. Malware Analysis with Images 30
2.4.1. Image Visualization 30
2.4.2. Classification Process 31
2.4.3. Results Obtained 31
3. Analysis **32**
3.1. Datasets Analysis 32
3.1.1. M0droid Dataset 32
3.1.2. Malgenome Dataset 34
3.2. Image Descriptors 41
3.2.1. GIST 41
3.2.2. Histogram 41
3.2.3. Image to Graph 42
3.2.4. Daisy 42

3.3. Features Selection 44
3.3.1. PCA 44
3.4. Machine Learning Classifiers 46
3.4.1. KNN 46
3.4.2. Naive Bayes 48
3.4.3. Decision Trees 49
3.4.4. Random Forest 50
3.5. Image Subtraction 51
3.6. Requirements 53
4. Design **55**
4.1. Dataset Preparation 55
4.1.1. Dataset Adjustment 55
4.1.2. Dataset Designing Decisions 58
4.2. Malware Transformation to Image 59
4.3. Classification Lines 59
4.3.1. Classification with Image Subtraction 59
4.3.2. Features Extraction and Classification 61
4.4. Programming Language Determination 62
4.4.1. Why Python? 62
4.4.2. Comparison with Another Languages 63
5. Implementation **65**
5.1. Project Environment 65
5.2. Libraries Used 65
5.3. Project Implementation Steps 66
5.3.1. Unpacking APK Files 66
5.3.2. Image Conversion 67
5.3.3. Subtraction Classification 71
5.3.4. Features Extraction and Classification 74
5.4. Problems Found 77
6. Performance Evaluation **78**
6.1. Tests Description 78

6.2. Accuracy 81
6.2.1. Subtract Classification 81
6.2.2. Extract Features and Classification 81
6.3. Time 83
6.3.1. Subtract Classification 83
6.3.2. Extract Features and Classification 83
6.4. Analysis of Results and Classifier Decision 87
6.4.1. Evaluation of Results 87
6.4.2. Final Decision 91
6.5. Final Results 92
6.5.1. Classification with M0droid Dataset 92
6.5.2. Packed Applications Classification 93
7. Project Design and Budget 94
7.1. Gantt Chart 94
7.2. Estimated Costs 95
7.2.1. Hardware Equipment 95
7.2.2. Software Licenses 95
7.2.3. Human Resources 96
7.2.4. Direct Costs 96
7.2.5. Indirect Costs 96
7.2.6. Benefits 97
7.2.7. Risks 97
7.2.8. Grand Total 97
8. Conclusions and Future Work 98
9. References 99
Annex I - Code 111
I.I. Unpacking the Application 111
I.II. Transform classes.dex into PNG Image 112
I.III. Subtraction Classification 113
I.IV. Extracting Features and Machine Learning Classification 114
Annex II - Confusion Matrices Images 117

II. I. Subtraction Classification 117

II. II. Extracting Features + Classification 118

Annex III - Time Measurement Table 134

Figures

Figure 1. Worldwide Smartphones Shipments 17
Figure 2. Worldwide Smartphone OS Shipments 18
Figure 3. Android Mobile Malware Families 18
Figure 4. Android Mobile Malware Variants 18
Figure 5. Image Conversion Process 30
Figure 6. Sections of the Resulting Image 30
Figure 7. Gist Image Descriptor 41
Figure 8. Histogram of an Image 41
Figure 9. Image to Graph Descriptor 42
Figure 10. Image with Daisy Descriptor 42
Figure 11. Daisy Descriptor 43
Figure 12. PCA Features Selector 45
Figure 13. KNN Classification 46
Figure 14. KNN Face Recognition 47
Figure 15. KNN Age Estimation 47
Figure 16. Naive Bayes Image Processing 48
Figure 17. Decision Trees in Sonography Analysis 49
Figure 18. Random Forest Face Recognition 50
Figure 19. Digital Image 51
Figure 20. Image Subtraction Process 51
Figure 21. Image Subtraction 52
Figure 22. Classification with Image Subtraction 60
Figure 23. Features Extraction and Classification 62
Figure 24. Goodware and Malware Image Examples (M0droid) 68
Figure 25. Different Families Image Examples (Malgenome) 69
Figure 26. ADR Examples (Malgenome) 70
Figure 27. Packed Applications Images 70
Figure 28. Confusion Matrix Subtraction Classification 73
Figure 29. Confusion Matrix Features Extraction + Classification 76

Figure 30. Accuracy Graph Features Extraction + Classification 82
Figure 31. Time Graph Features Extraction + KNN 83
Figure 32. Time Graph Features Extraction + Naive Bayes 84
Figure 33. Time Graph Features Extraction + Decision Tree 85
Figure 34. Time Graph Features Extraction + Random Forest 86
Figure 35. Best Classification (Daisy + KNN) Confusion Matrix 88
Figure 36. Worst Classification (Subtract with 1) Confusion Matrix 88
Figure 37. Confusion Matrix M0droid Dataset 92
Figure 38. Packed Applications Confusion Matrices 93
Figure 39. Gantt Chart 94
Figure 40. extract_F.py 111
Figure 41. extract_GM.py 111
Figure 42. convert2image.py 112
Figure 43. Malgenome APK to PNG 112
Figure 44. M0droid APK to PNG 112
Figure 45. sub_classif.py 113
Figure 46. classif.py (part 1) 114
Figure 47. classif.py (part 2) 115
Figure 48. classif.py (part 3) 116
Figure 49. Subtraction Classification Confusion Matrices 117
Figure 50. Gist + KNN 118
Figure 51. Gist + Gaussian Naive Bayes 119
Figure 52. Gist + Decision Tree 120
Figure 53. Gist + Random Forest 121
Figure 54. Histogram + KNN 122
Figure 55. Histogram + Gaussian Naive Bayes 123
Figure 56. Histogram + Decision Tree 124
Figure 57. Histogram + Random Forest 125
Figure 58. Image To Graph + KNN 126
Figure 59. Image To Graph + Gaussian Naive Bayes 127
Figure 60. Image To Graph + Decision Tree 128

Figure 61. Image To Graph + Random Forest 129
Figure 62. Daisy + KNN 130
Figure 63. Daisy + Gaussian Naive Bayes 131
Figure 64. Daisy + Decision Tree 132
Figure 65. Daisy + Random Forest 133

Tables

Table 1. Android Malware Detection Systems 29
Table 2. Correspondence between File Size and Image Width 30
Table 3. Requirement 01 - Number of Samples 53
Table 4. Requirement 02 - Computing Time 53
Table 5. Requirement 03 - Accuracy 54
Table 6. Original M0droid Dataset 55
Table 7. Original Malgenome Dataset 56
Table 8. Adjusted M0droid Dataset 57
Table 9. Adjusted Malgenome Dataset 58
Table 10. Subtraction Classification Tests 78
Table 11. Feature Extraction + Classification Tests 78
Table 12. Accuracy Features Extraction + Classification 82
Table 13. Time for 100,000 samples (hours) 91
Table 14. Budget - Hardware Equipment 95
Table 15. Budget - Software Licenses 95
Table 16. Budget - Human Resources 96
Table 17. Budget - Direct Costs 96
Table 18. Budget - Indirect Costs 96
Table 19. Budget - Risks 97
Table 20. Budget - Grand Total 97
Table 21. Time Measurement (seconds) 134

Glossary

- Malware: contraction of malicious software.
- PC: stands for Personal Computer.
- Exploit: attack on a computer system taking advantage of a vulnerability [132].
- Open-source: usually it refers to computer programs whose code is freely available [133].
- BSD License: stands for Berkeley Software Distribution and refers to free software licenses imposing minimal restrictions in the software distribution [134].
- MMS: stands for Multimedia Messaging Service referring to a standard way of sending multimedia content between mobile phones [135].
- P2P: stands for peer-to-peer and refers to the internet connection between peers (computers) forming a network where the peers can be a server or client, depending on the role they must have when sharing files [136].
- Jailbreak: action of gaining super-user access to an Apple Operating System (iOS) device [137].
- C&C: stands for Command and Control infrastructure where servers are able to control remote malware [138].
- URL: stands for Uniform Resource Locator which is an Internet addresses protocol [139].
- USB: stands for Universal Serial Bus which is a standard interface for connecting external peripherals to computers [140].
- Man-in-the-middle: attack where there is an individual which relays and modifies the communication between other two who are not aware of his presence [141].
- RBACA: stands for Role Based Access Control in Android [142].
- ASLR: stands for Address Space Layout Randomization and refers to a computer protection to buffer overflow attacks [143].
- ICC: stands for Inter-Component Communication [144].
- IMEI: stands for International Mobile Equipment Identity and refers to a code that identifies uniquely mobile phones [145].

- Rootkit: collection of programs which enables accessing with administrator privileges to a computer [146].

- IRM: stands for Inline Reference Monitoring which is a type of monitoring technique [147].

- UI: stands for User Interface.

- Pixel: smallest element of an image [148].

- IMSI: stands for International Mobile Subscriber Identity and refers to a code that identifies uniquely each user of a cellular network and it is integrated in the SIM (Subscriber Identity Module) [149].

- mTAN: stands for mobile Transaction Authentication Number that corresponds to the security number that banks send to clients via SMS to perform some operations [150].

- APK: stands for Android Application Package and refers to the package file format that is used in Android Operating System for mobile apps installation [151].

- DEX: stands for Dalvik Executable and refers to the executable file related with Dalvik Virtual Machine in Android [152].

- APN: stands for Access Point Name and refers to the gateway name of cellular and computer networks [153].

- ART: stands for Android RunTime and refers to the substitution applied in the new Android platforms between Dalvik and this new application runtime environment [154].

- Wi-Fi: technology used by computers to connect to a WLAN (wireless LAN) network [155].

- Bot: computer program that imitates human behaviors such as performing repeated operations [156].

- SDK: stands for Software Development Kit used for creating applications of a certain software package [157].

- Botnet: Internet connected computers that work autonomously and communicate between them via C&C [158].

- bpp: stands for bits per pixel [159].

- OOB: stands for Out Of the Bag technique applied in Random Forest classification [160].

- KNN: stands for K-Nearest Neighbors and refers to a type of machine learning classification [161].

- PNG: stands for Portable Network Graphics and refers to an image compression file format [162].

- OS: stands for Operative System.

- uint8: refers to unsigned 8 bit integer data type [163].

- Epoch: instant of time used as a reference for starting measuring time [164].

- PCA: stands for Principal Component Analysis used for selecting the most relevant features [165].

- HISTO: personal shortening of Histogram image descriptor used in this project.

- imgToGraph: personal shortening of Image to Graph image descriptor used in this project.

- F1 measure: also known as F1-score or F-score or F-measure, it is a measurement of the accuracy of a test, considering the precision and recall. The precision is the division between correct positive results and all the positive results. The recall is the division between correct positive results and the positive results that should have been retrieved [166].

1. Introduction

1.1. Purpose

The purpose of this project is an experimental analysis and development of a software alternative of Android malicious software (malware) classification, to be able to distinguish an application that contains malicious code with another that has not or to differentiate between families of Android malware.

1.2. Motivation

The smart devices usage has increased over the last years. Nowadays, smart devices refer not only to smartphones but also to smartTVs, smartwatches or tablets. They are extremely powerful and are designed with network and computing capabilities. However, most of them allow the user to obtain third-party applications, which will be run in the device, from a range of markets. This possibility means an insecure vulnerability that results in the intrusion of malicious software into the device, with the consequence of getting the personal data stored inside the device and accessing all the services provided by it.

Although there have been great progresses in PC malware detection, these advances has not been incorporate into smart devices. The majority of users that have a personal computer are aware of having a malware detection system, however this is not the case for smart devices; also this kind of users grant permissions to applications without enough privacy consideration or use unofficial markets to freely download applications that have a cost in official markets [1]. All of these aspects make really attractive for attackers to focus their efforts into smart devices.

According to a report made by *Business Insider*, the growth of smartphones shipments goes from around 400 millions in 2011 to 1,400 millions in 2015.[2] Furthermore, another study made by the same company shows that in 2014 the total number of Android devices bought were 1,042.7 millions from a total of 1,283.5 millions of smartphone shipments, which is 81.24% of all the smartphones sold. [3]

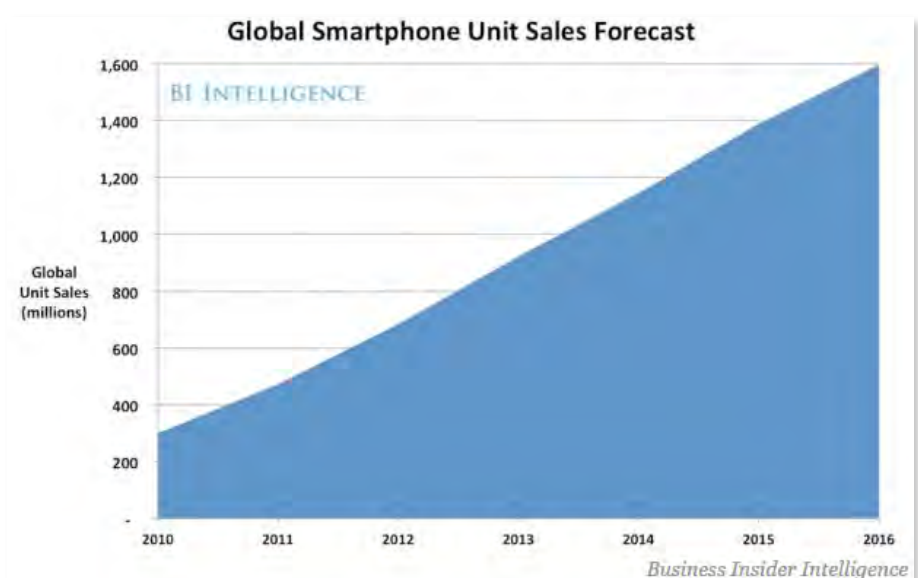


FIGURE 1. WORLDWIDE SMARTPHONES SHIPMENTS [2]

Global Smartphone OS Shipments (Millions of Units)	Q4 '13	2013	Q4 '14	2014
Android	227.3	780.8	291.7	1042.7
Apple iOS	51.0	153.4	74.5	192.7
Microsoft	9.6	35.8	11.3	38.8
Others	2.3	20.0	2.6	9.3
Total	290.2	990.0	380.1	1283.5

FIGURE 2. WORLDWIDE SMARTPHONE OS SHIPMENTS [3]

Due to this information, most of the malicious software is addressed to Android devices. Besides, Apple is known for its rigorous screening processes, which makes a laborious activity to try to endanger a non-jailbroken device. This is another reason why the number of malicious Android apps is so much bigger than iOS apps. One of the annual reports made by *Symantec*, an American technology company focused on computer security, shows that a total of 6% of new malware families appear in 2015 comparing with the growth of 2014 (20%). On the other hand, there was an increased of 40% of Android malware variants opposed the 29% of 2014. Moreover, the number of malware attacks against Android in the first quarter of 2015 was 550 attacks per day, increasing the volume of Android Malware by 230% compared with 2014. [4]

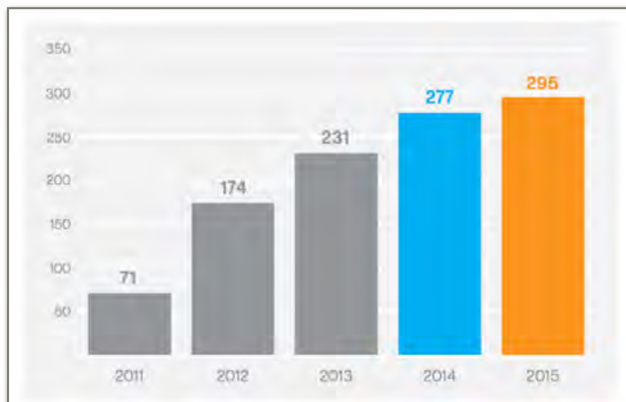


FIGURE 3. ANDROID MOBILE MALWARE FAMILIES [4]

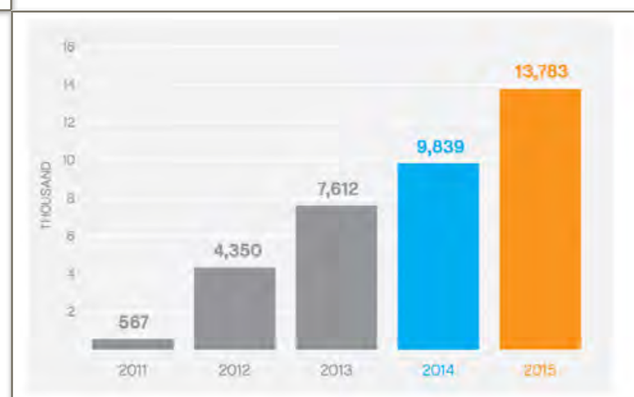


FIGURE 4. ANDROID MOBILE MALWARE VARIANTS [4]

Taking into account this data, the similarity between malware and the arrival of a new non-fully malicious software called grayware [5], which implies security risks the user is not conscious about, detecting efficiently fast different types of malware becomes a rough activity. Identifying malware rapidly with a high rate of accuracy is one of the objectives of cybersecurity currently. This fast detection can avoid unrepairable moral and economic damages. Thus the present thesis tries to classify accurately different types of malware within a reasonable time.

1.3. Scope

The scope of this project engages the following steps:

- Studying the evolution and characterization of Android malware, the security models used and its detections systems. Besides we will focus on the precedents about images evaluation with malware.
- Analyzing the two datasets used in the thesis development. Identifying and clarifying the malware families contained in the Malgenome dataset and the difference between malware and goodware of the M0droid one.
- Determining different machine learning features selectors and supervised classifiers that will be used in the grouping process of malware images. Additionally, a study about the subtraction of images will be carried out.
- Designing and implementing the image conversion and classification processes using the information of previous steps.
- Analysis of the results obtained from the final classification proposals.

Considering the study of the malware evolution, the detection systems and the security models, we will define them and expose how in Android is applied with some examples, as the project will be only focused in Android applications. The precedents of images evaluation will be explained considering malware addressed to personal computers and Android devices.

Since there are lots of families of malware, only the ones that appear in the two mentioned datasets provided are considered. As the project is based on analysis of malware through images, a deep study on each type of malware is not needed because at the end the importance falls on the image characterization independently of the role of this specific malware. Due to this, a brief definition of each malware family will be enough to know the main differences between them.

Regarding the machine learning classifiers, some of them will be picked up following a differential criterion in which the process of classification will be the one that determines the selection of it. The subtraction of images will be considered as one of the classification systems.

To test the different features extractors and classifiers, we will measure the time of extracting features, the training and testing times and the total time needed to run the whole program, as well as the accuracy.

Finally, all the results will be compared between them considering some accuracy and time boundaries in order to choose the best approach (features extraction and classifier).

Additionally, we have to comment that as it is an experimental analysis project, although we have developed some code to analyze the behavior, it is not a software development project so, in this way, we are not going to focus on the user interaction part that this kind of projects have to consider.

1.4. Structure of the report

This document describes each part of the project development as five sections:

- The first section, the *Introduction.* It comprehends the project organization as well as the motivation and purpose that bring the project to life.
- The second section, the *State of Art*. It refers to the first stage of the scope. It contains an investigation regarding the evolution and characterization of malware in smart devices, the security models, the malware detection and the malware analysis using images.
- The third section, the *Analysis*. It is related with the second and third stage of the scope. It contains an examination of the malware used and gathers a set of image descriptors, selector of characteristics, different supervised classifiers and the way of subtracting images. Additionally, some requirements are described.
- The fourth section, the *Design* process. It concerns the fourth stage of the scope. It uses the information obtained from the *Analysis* phase and exposes the study lines that will be develop in the next section, as well as the programming language in which the project is going to be implemented and tested.
- The fifth section, the *Implementation* phase. It refers to the fourth stage of the scope, as well. It shows the whole process of malware classification and the way it has been done, from the conversion into images to obtaining all of them classified. Besides, the problems encountered during this process are explained.
- The sixth section, *Performance evaluation*. It is related with the fifth stage of the scope. After performing some tests, the results will be shown graphically and the selection of the best one will be performed.
- The seventh section, *Project design and budget*. It shows the scheduling carried out in this project and estimates the possible costs that it will produce.
- The eight section, *Conclusions and future lines*. General and personal conclusions extracted from the development of the project and future work on this topic.
- The ninth section, *References*. It cites all the bibliography references used in the development of the project.

Additionally, three appendices are added:

- Appendix I - *Code*. It contains all the programming codes used during the development of the whole project.
- Appendix II - *Confusion matrices images*. In this section we include all the images of the confusion matrices obtained after performing the tests.
- Appendix III - *Time Measurement Table*. It contains the table which includes the time measurements of the different classifiers used in the project.

2. State of art

2.1. Malware Evolution and Classification

In this section, we propose the evolution of malware in smart devices (specifically to smartphones) from the initial disturbing apps to the ones designed to get some benefits like user information, economic gains or spying.
In addition, we show how malware is characterized and some Android examples, to use this knowledge in the classification process.

2.1.1. Malware Evolution

The early attacks made through smart devices were very similar to the malicious software oriented to PCs. In the 2000s *Palm* devices were infected with Trojans or classical virus like *Liberty* or *Phage*. The main goal of this malware was corrupting system files or damaging user information.

As the evolution in the mobile phone included new types of connections and communications like: SMS/MMS, Bluetooth, Wi-Fi, 3G, 4G and NFC [6] [7]; new malware strategies appeared. One early example was the worm called *Cabir* that affected *Symbian OS* devices using Bluetooth connections. When the Internet connection using these devices was possible, malware like *Yxes,* the first mobile botnet, were spread using SMS, affecting *Symbian* devices too.

The introduction of new mobile operating systems, paying services and mobile networking into smart devices caused an increase of malware addressed to them. Some examples, oriented to Android, are *NickiSpy*[1] (spies the device) or *Fakemini*[2] (sends premium SMS being the user unaware) or *FindAndCall* (used to steal sensitive information stored in an *iPhone* to leak it to a remote server connected to the network).

Finally, new smart devices appeared so new malware focused on them are generated. For example some Samsung smartTVs [8] have buffer-overflow vulnerabilities that could allow an attacker to monitor the device.

2.1.2. Malware Classification and Android examples

The classification of malware in old devices was based on the malicious objectives like corrupting files or spying (rootkits, spyware, trojan horses, adware, etc) or the propagations strategies like self-replication among others (worms vs. virus).

Current classifications are based on other terms [1]:

a) **Attack Goals and Behavior**: Some malware functions can be *monitoring*, *data exfiltration*, *sabotage* (removing critical files, draining the battery, etc), *spamming* (email, calls, SMS) and *fraud*.

[1,2] NickySpy and Fakemini are malware examples used in this project analysis (belongs to the Malgenome dataset)

In Android we can distinguish two different groups: botnets and grayware.
The first one is used for command and controlling (C&C) like *Andbot* [9](uses URL Flux) or *AnserverBot*[3] [10](uses obfuscation to hide URLs, server names and files, also the information published in blogs to keep in touch the members of the botnet and to update the code). Grayware collects and uses information store in a device without the user being conscious. For example, *Twitter* has been criticized for sending user phone contact list without been notified.

b) **Distribution and Infection:** There are six distribution strategies that can be used to infect smart devices:
- Market to Device: this technique is set up on market-borne attacks. For example, *DroidKungFu3*[4] [11] uses ingenious exploitation to be uploaded into the market without security detection and therefore, being installed in user devices.
- Application to Device: this technique is set up on application-borne attacks; they use another application, usually goodware, to replicate itself. For example, *Opfake-C* [12] uses *Facebook* to include malware inside the link posted.
- Web-browser to Device: this technique is set up on web-borne attacks; they are similar to the previous strategy but they use the Web instead of an application.
- SMS to Device: this technique uses SMS/MMS to be replicated.
- Network to Device: this technique is set up on device vulnerabilities. There are two types of distributions which follow this strategy: Cloud to Device (a powerful computer like a server or workstation is used in the distribution process) and Device to Device (another device is used in the distribution driven by P2P).
- USB to Device: this technique is based on the usage of USB port to spread malware in the device.

Apart from the examples mentioned before, there are two remarkably cases: repackaging, which consists on popular applications, usually with a cost in official markets, that are repacked with malware included and uses Market to Device distribution in alternative stores. One example is *FakeToken* (a man-in-the-middle attack). A repackaging variant is the other example, instead of including all the malware inside the repackaged app, it includes few lines of malicious code that allow the application to automatically download new malware once it is installed in a device (Network to Device distribution), usually with update looking.[13] For example, DroidKungFuUpdate[5] (update attack).

c) **Privilege acquisition**: most of the applications need some permissions to be executed so, there are two strategies used to grant these permissions:
- Technical exploitation: it takes advantage of misconfigurations or technical vulnerabilities. Some attacks performed using this technique include: API and System vulnerabilities, buffer overflows, rooted device vulnerabilities, etc. [14]
- User manipulation: it manipulates the user to obtain the permissions needed. Some techniques are: social engineering, repacked applications in unofficial markets and the inexperience in technical concerns of the user.

One example based on privilege acquisition is the rootkits that hide the malicious code from the operative system, Android *DroidKungFu*[6] variants hide rooting tools by modifying the name of the files exploited. [15]

[3,4,5,6] AnserverBot, DroidKungFu3, DroidKungFuUpdate and another variants like DroidKungFu1, DroidKungFu2, DroidKungFu4 and DroidKungFuSapp are used in the project analysis (Malgenome dataset)

2.2. Security Mechanisms

Android and iOS are operating systems originated to be used in smart devices. Although they inherit some security characteristics from the operating systems where they are built, there are another security features that are designed specifically for them. In this section we are going to focus on Android security mechanisms and then we will show a general overview of detection systems in smart devices.

2.2.1. Security Mechanisms Overview

We can distinguish between 3 main protection strategies:

a) **Market Protection**: it is the first protection wall malware has to deal with, it tries to avoid that malware reaches the official apps distribution markets. To afford this, there are two security mechanisms:
   - Application review: some markets examine the apps uploaded before changing their state to get ready to be downloaded and being available for the users. Android market is considered as an open-market against Apple market which is considered as a walled-market because it applies more exhaustive revisions techniques.
   - Application signing: applications authors are compelled to sign their developed apps. In this way, they assure the authorship and the integrity of the app, the device can verify the corresponding signature with the associated certificate authority.

   However, most of the researches [1] show that these mechanisms are not enough to avoid malicious applications. Reviewing each application manually, taking into account that there is a high number of applications uploaded every day, becomes a time consuming and laborious task.

   Considering hypothetically that the market protection techniques were perfect, there are a lot of users, as we commented before, that download applications from alternative markets in which there is not any security revision process and usually there is a fake signing, especially when the applications are supposedly a free copy of an application with a price in the official market.

b) **Platform Protection**: most of the smart devices platforms have a protection line to delimit the possible consequences of the malware installed in the device. There are four mechanisms that this platforms usually include:
   - Sandboxing: technique that is based on a trusted environment that uses an access control policy to isolate the applications that are running. This provides protection until some point because it does not avoid exploiting kernel vulnerabilities[16] and the case of user ignorance of app permissions.
   - Permissions: as we commented previously, most of applications require some permissions, with this the system restricts the apps behavior because its activity is bounded to a certain number of operations. However, as we see, the user generally does not take care of the permissions granted so an application, actually, can require more permissions that the ones really needed. A research made by Felt et al. [17][18] question the efficiency of this platform protection.
   - Interaction between apps: interaction between apps is normally based on components reusing but they can result on activity hijacking, system broadcast, broadcast theft and other malicious activities [19].

- Remote management: some platform manufacturers, network operators and markets are allowed to control apps remotely with the aim of updating them, applying patches, removing some technical errors or repairing damages in the device. However, this possibility can be used by attackers or in lot of cases, it is considered as a privacy intrusion by the users.

c) **New security mechanisms**: considering the limitations explained in the two previous approaches, it has appeared new ways of device protection. The most interesting ones are:
   - Access-control: lots of researches found as a possible protection mechanism different access-control techniques to allow the device to be shared between more than one user. Some examples are RBACA (Role Based Access Control for Android)[20] and DifUser[21].
   - Information-flow: high-level protection mechanisms believed in information flow in the system. Some examples are based on isolation with different profiles [22] and others on labeling systems [23].
   - Platform hardening: tries to simplify platform layers like the kernel or bootloader to avoid vulnerability risks. [24]
   - Rule driven policy: it proposes rules to improve policy languages. [25]

2.2.2. Security Mechanisms in Android

Android considers that users can choose freely which market they want to use to download the applications so, instead of market protection, it uses **platform protection**. The applications have the permissions in the manifest and if the user grants these permissions, the operating system will carry out with the execution of them at running time.

Regarding the platform protection system, Android uses **sandboxing** mechanism and another technique called **ASLR** (Address Space Layout Randomization). Android separates each app from the rest of running processes but these applications can establish a communication between them via ICC (Inter-Component Communication) that has vulnerabilities, as well.

In addition, Android forces that all the applications have to be signed using a **certificate** to guarantee developer's identification. However, this certificate can be no-verified by a certificate authority in the case of self-signing the app, so the developer is not properly identified and can be a fake certificate.

2.3. Malware Detection and Analysis

In this section we are going to explain the main strategies used in malware detection and the current detection systems oriented to Android. Finally, we propose a table in which we include some examples of Android techniques adopted to detect malicious applications that use the previous strategies explained.

2.3.1. Detection Strategies

We identify six different techniques used to detect malicious code in smart devices:

a) **Type of Detection**: depending on how the analysis of the code is performed to detect the malware, there are two types of analysis systems:
- Static analysis: this strategy unpacks and disassembles (decompiles) the application to detect suspicious blocks of code or malicious strings. It is a fast mechanism that is widely used in preliminary analysis to identify malicious code. Static analysis is considered as an efficient technique in market protection but its main disadvantage is that it does not detect malicious code if it is obfuscated or distributed separately from the app.
- Dynamic analysis: this strategy is based on deploying and executing the application on a controlled device or an emulator to identify suspicious behaviors of the app evaluated. In lot of cases, it needs automated or human interaction to detect these malicious behaviors, due to some harmful activities only appear after that another events occur firstly. The main advantage of dynamic analysis is that it identifies malicious applications even though the code is obfuscated, as the application has to be run. However, this is also a drawback because it is based on the user interaction so, it uses random events to analyze what happens next; thus it may be the case that the action when the malware appears is not executed, causing that it is not going to be detected.

Generally, static analysis is used for market protection and dynamic analysis for device-oriented protection.

b) **Type of Monitoring**: the malware detection can be performed by analyzing some monitoring features like user or network activities, permissions, system calls, event logs, instructions and program traces. This type of detection is based on the idea that any kind of application (with good or malicious code) relies on the device sensors and system to perform its activity, so the device components involved in this activity have to be challenged by the app to work.

- Regarding the device components involved in the analysis we can distinguish the following ones:
  - Hardware: there are features like the battery, information of the device and the input/output hardware identifiers from which it can be extracted some data that shows the presence of malware in the device. Examples of this case are the battery level or the phone IMEI.
  - Sensors: there are some components that show the user context, for example: the accelerometer, gyroscope, touch-id sensors, microphone, GPS, camera, speakers, among others. The usage of these elements can be monitored to detect suspicious behavior that can come from malware.

  - System: the monitoring of system resources like memory, storage, processes, package management and scheduler is an indicator of malicious actions.
  - Communications: features like network usage or phone and internet calls and messages can be monitored in order to analyze if the device is affected by malware, as they are a popular infection targets.
  - User: features that required user interaction like permissions, built-in and third-party apps are also indicators of malicious behaviors that can be monitored.
- The monitoring process can be accomplished at three levels:
  - Application-only: the monitoring is mainly focused on a specific application so there is an independent analysis for each application of the device. It is effective if the malicious software is a stand-alone app.
  - Group of applications: the analysis is done using the data gathered from a group of applications. Its effectiveness has been proven in the cases that malware is centered on apps collaboration in a distributed way.
  - Whole device: the analysis is performed considering a device monitoring instead of analyzing apps particularly. In rootkits cases, this monitoring method results very effective.

The main drawback of monitoring strategies is that it is a heavy resource consumption task, so it is important to select the proper monitoring strategy as well as the type of features to be analyzed, in order to reduce this consumption as much as possible.

c) **Type of Analysis**: The information obtained after monitoring is analyzed to obtain an evidence of the presence of malware in the device. Some strategies are clustering, data and control flow graphs, support vector machines, machine learning algorithms, self-organizing maps, etc.

d) **Type of Identification**: Once the data has been analyzed, there are techniques that identify the malware:
  - Anomaly-based: this strategy is based on the comparison of a predefined state, considered as “normal”, with the information obtained in previous steps. If there is any difference in the comparison, it is considered as suspicious or malicious. With this technique, malware that has not been seen before can be detected; on the contrary, it causes a lot of false positives because, as we explained, any rare behavior is malicious, when in real life it could not be the case.
  - Misuse-based: the identification of malware is carried out considering predefined patterns of signatures, usually for each known malware there is more than one signature. It is very accurate in identifying well-known malware. However, maintaining an up-to-date signatures database is a difficult task and the devices which few resources find really hard processing a huge amount of signatures.
  - Specification-based: this technique is based on predefined authorized specifications (behaviors), any difference between these specifications and the real behavior of the device is considered as malicious.

e) **Place of Monitoring and Identification**: monitoring and identification processes are resource consumption tasks, so the devices that are battery-constrained cannot afford them. In that way, there are three different places where these processes can be performed:
  - In the device: the processes are placed locally, so they must be lightweight and with a limited scope. There are two types of device-placed approaches: local

out-line, the device is monitored by the installation of itself in one of the device lower layers, this action requires root privileges; and local in-line or IRM (Inline Reference Monitoring), it is embedded in the app and does not need root privileges because it rewrites non-trusted apps.
- Distributed among other devices: these processes are placed in a cooperative way between different devices.
- In the Cloud: these processes are performed in virtual environments on a single server machine, the battery life is not reduced. There are two approaches: sandbox (uses dynamic analysis with controlled resources over the apps) and replica in the cloud (uses replicas of the device in remote secure servers).

f) **Place of Analysis**: depending on where the analysis phase takes place, we can distinguish two types of approaches: local and in the cloud. The local approaches obtain a response quickly; on the other hand, the cloud approaches have to wait for the results transmitted from the local preprocessing of the monitoring phase, this can take lot of time when the connection is slow or the transmitted packages are huge.

2.3.2. Contemporary Detection Systems

Currently, there are three categories for smart devices detection systems:

a) **Device monitoring systems**: most of these techniques use dynamic analysis, although some of them combine the two types of analysis (static and dynamic) to improve the malware detection. Device monitoring systems use as identification strategies: anomaly, misuse and replicas in the Cloud techniques.
- Anomaly Detectors: They are considered as dynamic analysis detectors and monitor features like the CPU usage and RAM memory. The analysis phase is done using machine learning algorithms like Tree Kernels, Self-Organizing Maps, Artificial Immune Systems and Support Vector Machines. This analysis is performed in the cloud. Some examples are: *Andromaly* [28], *TStructDroid* [29], *MADAM* [30] and *CrowDroid* [31].
- Misuse Detectors: the applications are usually repacked and the monitoring process follows IRM strategy, so it is embedded inside the applications. Applications can track themselves and define security permissions at runtime considering some security policies. The most common example is *AppGuard* [32].
- Replicas in Cloud: the monitoring, analysis and identification phases are performed in a secure environment with no battery problems. In addition, as several replicas can be executed simultaneously, the detection can be performed using different techniques at the same time. Some characteristic examples are *SeCloud* [33] and *Paranoid Android* [34].

b) **Automatic app-review systems**: These are used mainly in market protection considering virtual environments. The monitoring process in these environments has advantages like enabling virtual machine introspection to detect operating system semantics [35], an intensive analysis of security resources, and hosting in the cloud replicas of the device. There are four types of automatic app-review detection systems:
- Sandboxing: there are lots of approaches like *AASandbox* [36] or *DroidScope* [37] that use as detection technique the sandboxing strategy. However,

sandboxing limits the possibility to detect malware in a non-automated way because the user cannot interact with the application and some malware only appears after that.

- Smart Interaction: to solve the sandboxing problem mentioned before, it appeared a new technique that considers the activity made through the User Interface (UI). Two examples are based on this: *AppPlayGround* [38] and *SmartDroid* [39].
- Risk Analysis: these systems try to solve the threat communication problems between the usage of permissions and the user by a proof-of-concept instead of sophistication. The best example of this strategy is *RiskRanker* [40].
- Similarity Detection: to detect repackaging sources some strategies are focused on detecting correspondences between a set of applications. The first systems developed in this sense used syntactic analysis (string-based similarities) but the new ones are based on semantic analysis or other approaches like adapting the Vector Space Models. One example of this is *Dendroid* [41]. They are more expensive tasks than syntactic analysis.

c) **Attack-specific malware identification systems**: The two previous categories focus on general features to be detected, however due to the exponential growth of malware, another approach based on detecting specific malware characteristics has arrived. We can divide this approach into three:
- Privilege escalation: they are based on system vulnerabilities and leaking of inter-process capabilities. An example of the first case is the one explained by Checkoway et al. [42] and for inter-process capabilities some references are *CHEX* [43] and *WoodPecker* [44].
- Battery-depletion: they are based on energy-depletion attacks, usually the attacks affect the battery of the device. One example is *eDoctor* [45].
- Grayware: as we commented in previous sections, nowadays grayware is also a significant security problem. For this reason, it has appeared some approaches that try to detect privacy leakages due to grayware. Some examples are: *AppProfiler* [46] and *TaintDroid* [47].

2.3.3. Detection System Android Examples

Some examples mentioned in the previous section are analyzed deeply, considering all the issues explained before, and presented in the following table:

	Detection	Monitoring	Analysis	Identification	Place Monitoring & Identification	Place Analysis	Consumption	Features monitored	Observations
Andromaly	Dynamic	ALL	Machine Learning	Anomaly	Local-outline	Local-outline	- RAM: 8.8% - CPU: 5.52% - Battery: 10%	- CPU consumption - Network packages - Running processes	- Labelled data classification (training). - Anomaly attacks - Accuracy: 40%-100%
TStructDroid	Dynamic	Process Control Block	- Machine Learning - Statistical	Anomaly	Local-outline	Local-outline	Performance degradation: 3,73%	- Page frames - Page faults - Context switches - Virtual memory	- Any kind of attacks - Accuracy: 98%
MADAM	Dynamic	- User Level - Kernel Level	Machine Learning	Anomaly	Local-outline	Local-outline	- Memory: 3% - CPU: 7% - Battery: 5%	- Kernel features: System calls, memory, CPU - User features: key strokes, user state, phone calls, SMS, NET	- Anomaly attacks - Accuracy: 93% (KNN - K=1)

CrowDroid	Dynamic	System calls	Clustering	Anomaly	Local-outline	Cloud	*Not available*	Apps system calls	- Anomaly attacks - Accuracy: 85%-100% (K-Means)
AppGuard	Dynamic	Program traces	*Not available*	Misuse	IRM	Cloud	Performance degradation: 5%	- Generated events - Program traces	- Privacy leaks attacks - Analysis is done off-line
SeCloud	- Dynamic - Static	ALL	ALL	- Anomaly - Misuse - Specification	Replica in the Cloud	Cloud	*Not available*	Any kind of features	Any kind of attacks
Paranoid Android	Dynamic	ALL	ALL	- Anomaly - Misuse - Specification	Replica in the Cloud	Cloud	Performance: 64B/s - 2KB/s	*Not available*	Any kind of attacks
AASandbox	Dynamic	ALL	Clustering	Misuse	SandBox	Cloud	*Not applicable*	*Not available*	Any kind of attacks
DroidScope	Dynamic	ALL	*Not available*	*Not available*	SandBox	Cloud	*Not applicable*	Any kind of features	Any kind of attacks
AppPlayGround	Dynamic	- System calls - Program traces	*Not available*	*Not available*	Cloud	Cloud	*Not applicable*	Any kind of features	Heuristic-based UI interaction
SmartDroid	- Dynamic - Static	ALL	ALL	- Anomaly - Misuse - Specification	SandBox	SandBox	*Not available*	Any kind of features	- Hybrid dynamic and static detection - UI attacks
RiskRanker	Static	- Instructions - Permissions - API calls	Dependency Graphs	Misuse	Cloud	Cloud	*Not applicable*	- Permissions - API calls - Vulnerability signatures	- Any kin of attacks - Ranks the severity of suspicious operations
DenDroid	Static	Instructions	- Dependency Graphs - Clustering - Statistical	*Not available*	Cloud	Cloud	Performance degradation with big datasets	Code chunks	- Detects unkown malware - Text mining and information retrieval classification
CHEX	Static	Instructions	Dependency Graphs	*Not available*	Cloud	Cloud	*Not applicable*	User data	User private information attacks
WoodPecker	- Static - Dynamic	- Instructions - Permissions	Dependency Graphs	*Not available*	Cloud	Cloud	1 hour detection	Explicit and implicit leakages	- Permissions analysis for implicit leakages - CFG for explicit leaks
eDoctor	Dynamic	Hardware	Clustering	Not available	Local-outline	Local-outline	Battery: 1.5%	- Resource usage - Energy consumption - Events	- Accuracy: 94% (k-means)
AppProfiler	- Static - Dynamic	- API calls - Program Traces	Expert	Machine Learning	Local-outline	- Local-outline - Cloud	*Not available*	- API calls - Permissions	- Privacy leaks attacks
TaintDroid	Dynamic	Program Traces	Expert	Machine Learning	Local-outline	Local-outline	- Memory: 4.4% - CPU: 14%	- Variables - Methods - Messages - Files	- Explicit information attacks

Table 1. Android Malware Detection Systems

2.4. Malware Analysis with Images

As this project is based on analyzing malware with images, the precedent regarding this topic is a research study made, in 2011, by Nataraj et al. [49] with malware targeted to Windows computers. In December 2015, they extend this study to Linux, Mac OS X and Android platforms [128].

These researches classify malware using image processing techniques. They obtained gray-scale images from the malware binaries. The study made in 2015 is a comparison between platforms, applying the same strategy used previously in 2011.

2.4.1. Image Visualization

To visualize the gray-scale images from the malware binaries, a process of reading these files as a vector of 8 bit unsigned integers and rearranging them into a 2 dimensional array is performed. The pixels' values of the images will vary between 0 and 255, being 0 a black pixel and 255 a white pixel, its width is fixed depending on the binary file size and the height will differ depending the width and file size.

Depending on the size of the file, the corresponding widths are summarized in the following table:

File size Range	<10KB	10KB - 30KB	30KB - 60KB	60KB - 100KB	100KB - 2000KB	200KB - 500KB	500KB - 1000KB	>1000KB
Image Width	32	64	128	256	384	512	768	1024

Table 2. Correspondence between File Size and Image Width

Graphically, the image conversion process is something like this:

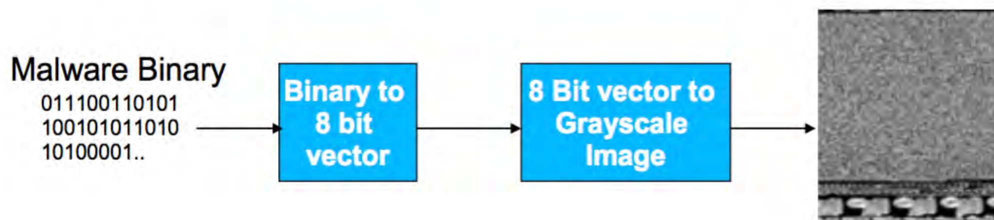
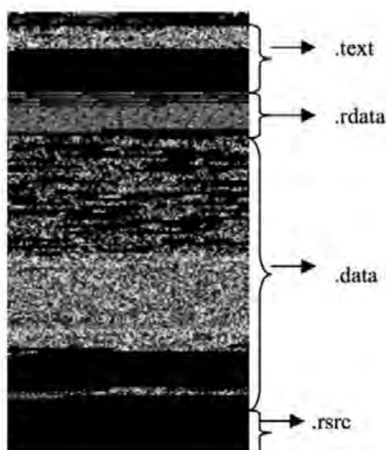


FIGURE 5. IMAGE CONVERSION PROCESS [49]



After the conversion of the binary files into images, the resulting gray-scale images are shown in the following way:
The executable code is contained in the upper-side of the image (*.text*). Here we can see two different parts: the part with code (fine grained) and the part with zero-padding at the end (black).
In the middle of the image it contains the data initialized, which corresponds with the textured part and the non-initialized data represented as black.
Finally, the resources of the file, like icons the program may use, are at the end of the image (*.rsrc*).

FIGURE 6. SECTIONS OF THE RESULTING IMAGE [49]

2.4.2. Classification Process

The classification process is divided in two parts:

a) **Extracting images features**: There are three main areas based on analyzing images textures: texture classification (identifies uniform textures in an image), texture analysis (identifies boundaries of texture regions in an image) and texture synthesis (synthesizes the image textures).
Considering the textures of the images obtained, in this study they use GIST [50], which is based on wavelet image decomposition, with 320 features values. The average time to compute the GIST features values of an image is 54 ms.

b) **Cross-validation classification**: After obtaining the image features, they use k-nearest neighbors with Euclidean distance to do the supervised classification. The tests were performed with 10 fold cross validation. In each iteration, 90% of the images belonged to the training set (chosen randomly) and 10% to testing set. Therefore, a given test image is going to be classified to a class considering the modulus of its k-nearest neighbors.

2.4.3. Results Obtained

In the study made in 2011, the experiments were performed using a dataset of 9,458 Windows malware binaries, divided in 25 families, and varying the number of neighbors in a range of 1 to 10. The best accuracy was 97.18% with 3 neighbors.

As this project is focused on Android platform we are only going to present the results obtained in Android experiments of the research made in December 2015.

They used the same strategy than in 2011 with the Malgenome dataset applied in this project too. However they only considered 13 families with at least 20 samples each. The obtained accuracy was 84.55%.

In this project we are going to use as much families as possible because we want to analyze malware in a very realistic way. The results obtained in this project will be more adjusted to real cases as the number of families is not fixed and it can be families with little number of samples that must be considered. Besides, we are going to use more Android datasets.

3. Analysis

To develop this project, we have used two datasets. The first one contains malicious and normal applications (M0droid dataset) [92] and the second one contains different malware families (Malgenome Project dataset) [93].

In this section we are going to study the characterization of each dataset, we will analyze the process of extracting image features with some image descriptors and how to use the values obtained after this extraction, considering some machine learning classifiers and features selectors. Moreover, we will analyze the process of subtracting one image from other to check the differences between them, as this process is often used in photoshopped analysis or in motion detection and could be useful for this project.

3.1. Datasets Analysis

In order to understand the behavior of the classification process, and consequently to develop properly the present thesis, we have performed an analysis of the malware that appear in each dataset. This analysis is based on defining and explaining the main differences between goodware and malware and the characteristics of each malware family.

3.1.1. M0droid Dataset

This dataset is part of the M0droid Project, which dates from 2015 [92], so it is a new and contemporary dataset. Inside it we found some samples gathered in two groups: goodware and malware. Considering the definition provided by Symantec, malware can be defined as "*an abbreviated term meaning 'malicious software'. This is software that is specifically designed to gain access or damage a computer without the knowledge of the owner.*"[87]. In this case, they are oriented to Android devices. Therefore, the main difference between the two groups inside this dataset is that in the group with **goodware** applications, there is **no malicious software** and in the other one (**malware**), the applications contain **malicious code**.

Despite in this dataset the malware is not divided into different types, there are six **collections of malicious applications** where we can classify malware, depending on the behavior and objectives of them. We will give a brief explanation of each of them because we think that maybe inside the dataset there are different types of malware, so it is an important fact to consider in terms of their identification. Although current classifications techniques do not depend on the malware behavior, as we comment in the previous section (*2.1.2. Malware Classification and Android examples*), we believe that it is important to know the purposes why malware is created and also it is a way to understand the division into families:

- **Adware:** A malicious application can display advertisement inside it or by launching the Web browser of the device. An attacker can repackage an application including an advertisement library registered to himself. The repackaged application works as the original, so the user is not aware about using an illegitimate version. Besides, usually these applications use the location, the contact list, the IMEI and IMSI of the device to

identify the user and create personalized ads. Each time the ads are displayed, the attacker will obtain a revenue. [88]

- **InfoStealers**: applications that obtain confidential information from the device being the user unaware. This information is used for several purposes like spamming or accessing to social media. The information is usually sent to C&C remote servers [89].

- **Spyware**: applications that track the user device. They can monitor incoming/outgoing calls, user location, text messages, email and tracking user web browsing [88] [89].

- **SMS Trojans**: the attacker can obtain a revenue by creating a malware that sends SMS messages to premium numbers, making the user to pay an extra charge [89].

- **mTAN Stealers**: some banks require additional credentials, sent to the user via SMS, when login into an online bank account or making a transaction. These credentials are usually a random set of numbers, called mTAN (mobile Transaction Authentication Number) that can be intercepted with malicious applications [88] [89].

- **Ransomware**: malware that blocks the user to access to the device by locking the access to the system or encrypting the user files. It forces the user to pay a quantity of money to access again to the device or recover the data [90] [91].

Furthermore, even though most of Android applications are trojans [89], we can distinguish other types of Android malware as well:

- **Trojan horse**: computer program that is installed by the user because it does not seem dangerous (or is attached to another program) but its behavior is completely unexpected from the user, like encrypting the files or downloading and installing other malicious programs. It cannot replicate itself.

- **Virus**: malicious software that is hidden inside another apparently inoffensive application and has the ability of self-replicating (creating copies of itself). Mostly, it performs actions like changing or deleting data.

- **Worm**: malicious software that makes copies of itself, as viruses. The difference with viruses is that it is not attached to any other application.

- **Rootkit**: malicious application that has the ability to hide itself from the user and perform changes in the device or steal information. It commonly uses root privileges.

- **Backdoor**: malicious application that creates a bypassing procedure to authenticate or perform any action instead of using the normal methods, with user’s awareness.

Finally, as we are dealing with Android applications (APK files) we are going to explain the contents of an Android application. An APK file is an application container that has been packed through a zip algorithm and includes all the application resources and files. An unpacked application contains the following files and folders [97]:

- **AndroidManifest.xml**: it is a file that represents the configuration file of the application. It contains the application unique identifier, the permissions needed by the application to work or the components (<<*activities*>>, <<*content providers*>>, <<*receivers*>>, etc.) of the application [96]. Once the application is installed in a device, the *PackageManager* reads this file and sets up and deploys the application on the Android platform [97].

- **classes.dex**: file that contains the compiled code of the application in DEX (Dalvik Executable) format, in order to be interpreted by the Dalvik virtual machine or ART (Android Runtime). All the applications are executed in the virtual machine with security purposes [96].

- **resources.arsc**: it is a file that stores the compiled resources [96].

- **res**: it is a directory that contains all the resources used in the application such as icons, images, strings in several languages, tones, etc. [97]. Some applications may not contain this folder if they do not use any resource of this type.

- **assets**: it is a directory that contains resources that are not compiled (assets). They can be retrieved by the *AssetManager* [97]. This is also a not mandatory folder.

- **META-INF**: it is a directory that contains information regarding the digital signature of the application. This information is presented in three files [96]:
  - <u>MANIFEST.MF</u>: list of all the files, with their corresponding SHA-1 hash, inside the APK.
  - <u>CERT.SF</u>: SHA-1 hash of every 3 lines that are found in the *MANIFEST.MF* file.
  - <u>CERT.RSA</u>: it contains the signature of *CERT.SF*, so it stores the APK signature.

3.1.2. Malgenome Dataset

This dataset is part of the Malgenome Project, which starts on May 2012 [93]. This dataset contains 49 Android malware families. Now we are going to analyze each of them:

- **ADR** [51]: Trojan that comes from a legitimated application which has been modified and uploaded to the market. The main goal of this malicious application is stealing information from the device like the IMEI, IMSI, APN, Wi-Fi, network connectivity, hardware information, etc. These actions are performed when some of these events happen: a phone call is received by the device, it has passed twelve hours since OS started and the re-establishment of the network connectivity has been performed after being lost.

- **AnserverBot** [52]: it is a bot program, specifically it is considered a Trojan, distributed in alternative markets in China. This malicious app works following these steps:
  1) When it is launched, it will display a fake upgrade dialog that will download and install another bot program. This bot program will run in the background without the user being noticed. This bot program can run even when the AnserverBot has been removed from the device.

2) There is another bot program inside AnserverBot that is not installed in the device but it is launched dynamically and executed at runtime by AnserverBot directly or by the bot installed previously.
3) AnserverBot periodically (one time per two hours) communicates remotely with C&C servers to recover some commands or other information.
4) Some considerations of this malware are that it is protected from reverse-engineering using obfuscation and from being repackaged. It is considered more sophisticated bot program than *Pjapps* and *BaseBridge* and includes dynamic code loading like *Plankton*.

- **Asroot** [53]: malicious application that gains root access to avoid security sandboxing with user awareness.

- **BaseBridge** [54]: it is a Trojan malicious application that tries to send SMS to premium numbers.

- **BeanBot** [55]: Trojan included in repackaged third-party applications of alternative markets, that are supposedly free versions of paid apps. It communicates via C&C with remote servers and sends premium SMS.

- **Bgserv** [56]: Trojan malware that creates a back-door to send user information from the device to a remote location.

- **CoinPirate** [57]: Trojan app repackaged of a game named "*Coin Pirates*" that is available as a free application in Chinese markets. Its functionality is based on filtering text messages received, considering some keywords defined by the malware author. If a text message contains one of these keywords, it will be removed or uploaded to a remote server with information like the IMEI, IMSI, SDK version and device model.

- **CruseWin** [58]: Trojan that displays, in the application list of the device, a standard Flash icon that downloads a XML configuration file to obtain a set of URLs for sending and retrieving data. Besides this malicious application can delete SMS, send premium SMS to the numbers included in the XML downloaded previously, update itself or even delete itself.

- **DogWars** [59]: Trojan that sends SMS to all the members of the contact list stored in the device. It is a repackaged version of the application called "*Dog Wars*", available in third-party markets.

- **DroidCoupon** [60]: Trojan app that communicates via C&C with remote servers for sending the IMEI and subscriber ID and receiving instructions to remove or download and install other packages.

- **DroidDeluxe** [61]: root-exploit malware that takes device information like the manufacturer, brand and device model.

- **DroidDream** [62]: root-exploit malware that communicates via C&C to send information such as product identifier, partner, IMEI, IMSI, SDK version, language, country and the user identifier.

- **DroidDreamLight** [63]: malware that affects around 30,000 to 120,000 Android devices, it was contained in applications available in the official Android market. The author of this malware is believed to be the same that *DroidDream* or other *Droid* versions. Some of the applications that contain this malware were the ones develop by *Magic Photo Studio*, *Mango Studio*, *BeeGoo*, *E.T. Teen*, *GluMobi* and *DroidPlus*. Once these applications were installed, the malware stole information stored in the device like the list of the apps installed, IMEI, IMSI and SDK version. This information was uploaded to several URLs where the device connects to. Besides, with the user intervention, this malware can download and install new packages.

- **DroidKungFu1** [64]: this malware is included in repackaged apps in Chinese alternative markets. The malware includes inside these apps a receiver and a service. The service will be launched, being the user unaware, after the receiver is notified when the booting process has finished. The service is based on rooting the device by encrypting two known root exploits (*udev* and *RageAgainstTheCage*). The malware decrypts these two exploits and then executes them to send information (Android OS version, IMEI and phone model) to an URL. Once the root privilege is granted, it can install or access to any file, even remove them.

- **DroidKungFu2** [63]: improved version of *DroidKungFu* that, apart from root-exploiting the device, it communicates with three C&C URLs servers, stored in a native file, and reads device information, writes them to a local file and uploads it to one of these servers.

- **DroidKungFu3** [65]: new variation of *DroidKungFu* more advanced than the previous ones. The goal of this malware is avoiding being detected by mobile anti-virus software. It contains obfuscation mechanisms to hide the C&C URLs servers (previous versions contain them as plaintext), encrypts malware binaries and masquerades app as a Google Update.

- **DroidKungFu4** [66]: it is a version of *DroidKungFu3* but instead of encrypting C&C addresses in a Java class file, as it is done in *DroidKungFu3*, they are stored as ciphertext in a native file, like *DroidKungFu2*.

- **DroidKungFuUpdate** [67]: malware that uses the Update Attack. In this case, a prompt appears to the user with an update request after the installation. If the user accepts, it will download malware that belongs to *DroidKungFu3* family.

- **EndOfDay** [68]: Trojan that was embedded in a fake version of the "*Holy ***king Bible*" application. When the device reboots, a service called "*theword*" starts and periodically it tries to send the phone number and operator code to a host service. It attempts to receive information from a remote location in a period of 33 minutes, as well. Additionally, there were two activities programmed to the 21th and 22nd of May 2011. At that time, on 21th, it created a database called "*mydb.db*" in which it wrote some texts with the word "*endoftheworld*". Then, it selected randomly some of them and sent them to the entire contact list, it changed the wallpaper of the phone too. On 22nd of May 2011, it changed again the wallpaper and sent again an SMS to each member of the contact list with a new message.

- **FakeNetFlix** [69]: phishing application that is available in alternative markets. The app asks for Netflix username and password and sends them to a remote server. When the user introduces them, an incompatibility error screen appears and the credentials are sent.

- **FakePlayer** [63]: this is one of the earliest Trojan malware discovered in Android (2010). It attempts to send premium SMS to Russian numbers, having an appearance of a movie player the user has to install in the device.

- **GamblerSMS** [63]: it is a spyware application that monitors every incoming and outgoing SMS and records every outgoing phone call. It runs in the background and the information is sent to the attacker.

- **Geinimi** [63]: Trojan that is embedded in a legitimate repackaged application. After being installed, a backdoor is opened and information from the device such as geographic and contact details are sent to a remote location. Besides, it can send SMS or make a phone call to specific numbers, delete SMSs, download files, send and receive information to C&C servers (the network addresses are encrypted), change the wallpaper, install or delete packages, etc.

- **GGTracker** [63]: Trojan that sends and receives SMSs to premium numbers. It affects American users. It was distributed as a battery-saving application in third-party markets.

- **GingerMaster** [63]: first Android malware with root-exploit purposes. It was available in alternative Chinese markets. Once it is installed, a receiver is registered and notified when the reboots ends. After that, a service (that collects device information and sends it to a server) is launched and runs in the background. When the application has root privileges, it connects to C&C to receive instructions to install new APK files.

- **GoldDream** [70][71]: Trojan that spies received SMS and incoming and outgoing phone calls to send them later to a remote location, being the user unaware. It was spread in alternative Chinese markets embedded in games like "*Draw Slasher*" or "*Drag Racing*". In addition, it has bot capabilities (receive C&C instructions from a server to be executed).

- **Gone60** [72]: Trojan that was available in official Android market as: "*Gone in 60 Seconds*","*Gone in 60s*", "*gi60s - reveal secrets"*, "*gi60s - reveal secrets UN"* and *"Get secret data in 60 seconds*" whose publishers are *CLOUDDOG* and *CREATIVEDOGS*. It asks the user for the following permissions: reading contact data, sending SMS messages, opening network connections and reading Web browser bookmarks. It attempts to steal user information such as the user contact list, SMS messages, visited URLs and recent calls. It may encode and upload this data with Base64 encryption to this location: http://gi60s.com/upload.php.

- **GPSSMSSPy** [73][74]: malware that records and sends to a remote server the current location of the user and the SMS messages.

- **HippoSMS** [63]: Trojan that appears in Chinese third-party markets that sends SMS messages to premium numbers. It blocks numbers which can inform the users about the additional charges made to their accounts and removes numbers starting with 10 (in China legal mobile phone service's providers begin with 10).

- **JiFake** [63]: Trojan that sends premium SMS oriented to Russian users. It was included inside *Jimm*, a popular Russian ICQ app. It uses the QR system to be installed, so when the user scans the code, it will be redirected to a site that will install the Trojan.

- **jSMSHider** [75]: Trojan that has appeared in alternative Chinese markets and affects ROM or rooted devices. It receives commands from a remote server by installing a communication payload considering the exploitation of a ROM vulnerability. It can read and send SMS messages and install applications.

- **KMin** [76]: This Trojan sends to premium numbers data such as IMEI, IMSI and other files.

- **LoveTrap** [63]: Trojan repacked in legal applications like e-book readers and location tracker apps. It sends premium SMS messages and blocks messages that inform the user about this additional charge. Additionally, it sends system information to a remote server.

- **NickyBot** [77]: Spyware that is controlled by commands via SMS messages, so it is also a bot program. The main commands are:
  - *record*: Recording the sounds in the phone, this sound can be phone calls, the surrounding sounds, and any kind of sound.
  - *contact*: Sending to a predefined email address the contacts data in the phone.
  - *boot*: Enable/disable the functionality about booting notification, which will send a SMS reply after booting.
  - *log*: Enable/disable the monitoring of phone calls.
  - *sendlog*: Sending to a predefined email address the phone call logs.
  - *sms*: Enable/disable the monitoring of SMS messages.
  - *sendsms*: Sending to a predefined email address the SMS messages.
  - *gps*: Enable/disable the monitoring of GPS location.
  - *state*: Checking and reporting the monitoring status of all the possible functionalities (enabled/disabled).
  - *all*: Enable/disable all the functionalities that can be monitored.

- **NickySpy** [78]: Trojan program that spies the device and steals the information collected.

- **Pjapps** [63]: Trojan with backdoor capabilities, considered as a repackaged version of legal applications like "*Steamy Window*" (steamy window effect on the device screen). This malware attempts to build a botnet with C&C servers to send and receive messages to install new applications, visit Web sites, send and block messages and add bookmarks to the Web browser.

- **Plankton** [63]: Trojan that is repackaged in legal applications. It is also known as *Tonclank* and when it is executed, it will obtain and send the device identifier and permissions to a remote server. From this server, a *.jar* file is downloaded that will open a backdoor and, depending on the command received, it will create a registry of all the activities of the phone, copy the bookmarks, modify the homepage of the Web browser and return the status of the previous command.

- **RogueLemon** [79]: in China there are subscriptions services that work via SMS communication between the user and the service provider. There is a communication policy that controls the communication. However, RogueLemon violates this policy. It registers a SMS receiver in the manifest file of the application with the highest priority (99999) so in that way this receiver will be the first one that handles the incoming SMS. If the SMS is from a service provider, it will connect to a remote server and interchange some encrypted messages that contain user information. Then, it will answer with a positive confirmation to this service, the user will be charged and signed up to the service without knowing it.

- **RogueSPPush** [80]: it uses the same functionality that *RogueLemon* but in this case the messages are not encrypted.

- **SMSReplicator** [63]: Trojan available in 2010 at Android market with a price of $4.99. It was a spyware that sent the incoming SMS to a defined phone number or a Website like androidversion.net or criptosms.com. It was forbidden from the Android market a few hours later.

- **SndApps** [81]: this malicious software was included in the Android market and is a payload that once it starts, it will gather information like the phone number or email addresses stored in the device, and send them to a server remotely placed.

- **Spitmo** [82]: malware that once is installed, the bank incoming SMS are monitored and it steals mTAN (mobile Transaction Authentication Number) messages.

- **TapSnake** [83] [84]: Trojan embedded in a game similar to the "*Snake*" game. It steals the user location and sends it to a remote Web service.

- **WalkinWat** [63]: Trojan inside a repackaged application of a legitimate *Walk* app available in alternative markets. It steals all the user data and sends it to a remote server (whose domain was incorporateapps.com). Besides, it sends a SMS message to all the members of the contact list telling that the user has installed a fake version of the app. Moreover, it accesses network information,

vibration features, phone location, turning on and off the device. Finally, a prompt appears warning the user about not downloading fake apps and with two options: visiting official Android market or exiting the application.

- **YZHC** [85]: Trojan that sends premium SMS to numbers obtained from the Internet and removes this activity once it is performed. It was available in the official Android market during 4 months and is available in Chinese alternative markets.

- **zHash** [63]: root-exploiting malware that appeared in Chinese third-party markets. This was the same exploit than some versions of *DreamDroid.* If the rooting process was successful, other applications can also obtain root privileges with user awareness.

- **Zitmo** [63]: malware that intercepts bank information from SMS messages that is sent to remote server. In that way the attacker can use the user bank credentials to login the account and perform modifications.

- **Zsone** [86]: Trojan included inside 10 apps (*iMatch, 3D Cube horror terrible, ShakeBanger, Shake Break, Sea Ball, iMine, iCalendar, LoveBaby, iCartoon* and *iBook*), developed by *Zsone.* Once the user starts the application, it will silently send SMS messages to costly subscription services in China with user awareness. It was available in the official Android market and alternative markets.

3.2. Image Descriptors

In this section we are going to study different image descriptors to extract a features vector from the images obtained. We will start from the one used in previous malware image researches, GIST image descriptor, and then we will extend this study to others features extractors such as histogram, image to graph and daisy descriptors.

3.2.1. GIST

This descriptor was proposed in [94]. Firstly, it divides the image into segments forming a 4x4 grid. Secondly, it extracts the orientation histograms. The values obtained in the second phase constitute a vector of 960 values. Usually, before applying the GIST descriptor, the image has to be resized into a smaller squared image between 32 and 128 pixels. This is due to the fact that this descriptor does not represent the image details so it is considered as a low dimensional descriptor [95].

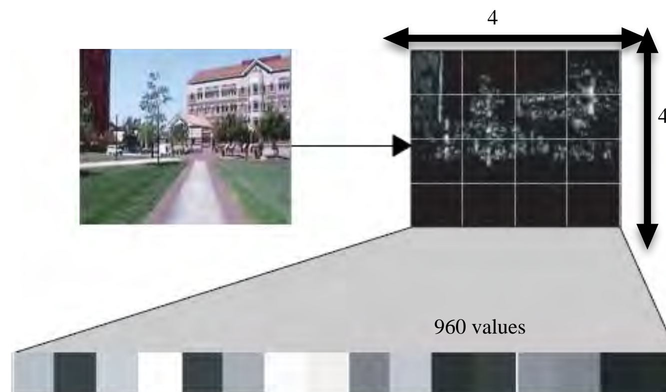
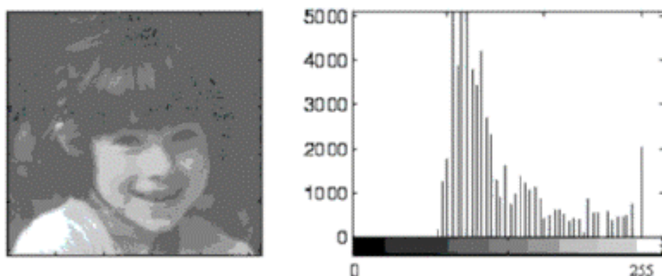


FIGURE 7. GIST IMAGE DESCRIPTOR [168]

3.2.2. Histogram

A histogram is a graph that shows the frequency of an object or instance. In case of images, it presents the frequency of the intensity levels of the image pixels. Particularly, the *x* and *y* axis of the graph represent the image gray-scale intensities and the frequency of them, respectively. [98]



As we can see from the figure, the *x* axis corresponds to the pixels values, it goes from 0 to 255 because the image processed is a 8 bbp (8-bits per pixel) image, so it means that there are 256 possible levels of grays (2^8 = 256 colors). On the other hand, the *y* axis contains the intensity quantities.

FIGURE 8. HISTOGRAM OF AN IMAGE[169]

From the image of *Figure 8*, we can conclude that it is a light image, which it can be proven just looking at the image, however the reasoning extracted from the histogram about why it is a lighter image is that most of the high frequencies lie in the middle of the graph or in the in the right half part, where corresponds to the high levels of gray colors. In this image there is not a dark side because there are not values represent in the graph.

As the operations needed to perform a histogram are very basic, we think that this descriptor could be a fast descriptor that can be useful in this project.

3.2.3. Image to Graph

A descriptor developed by *scikit-learn* is *img_to_graph* [99]. This descriptor studies the connections between neighboring pixels of an image. It analyzes pixel-to-pixel gradient connections and draws a graph from the connectivity matrix built from the image. This matrix establishes the connections between the pixels, so that the edges of the image are weighted with the gradient values. We will obtain the different values retrieved from this descriptor considering the graph obtained.

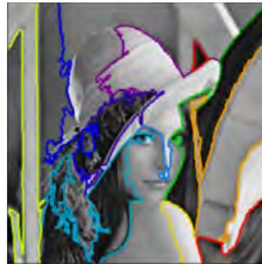


FIGURE 9. IMAGE TO GRAPH DESCRIPTOR [99]

As we can see from *Figure 9*, this descriptor detects perfectly the different color variations and therefore, the edges of an image, so it could be useful in our experiment because we want a descriptor that distinguish clearly each part of the image (.text, .data, etc.) and assign a value to each part.

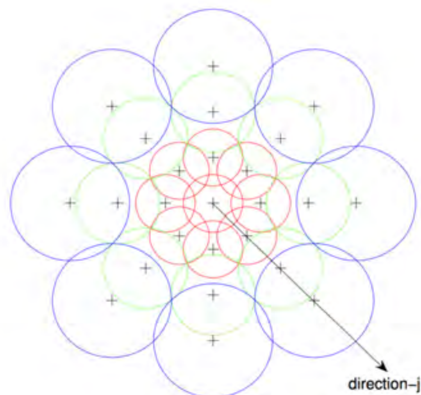
3.2.4. Daisy

Lot of researches [101] [102] [103] considers SIFT (Scale-invariant Feature Transform) and its variants a good image descriptor. The same thing happens with BOF (Bag of Features) descriptor [104] [105] [106]. A similar approach that considers the benefits provided from the SIFT and BOF is DAISY descriptor.

It is a more computational efficient descriptor that is based on the gradient orientation histograms in a similar way than SIFT descriptor does [100] [107]. The way it is formulated allows a fast dense extraction, like BOF strategies.

FIGURE 10. IMAGE WITH DAISY DESCRIPTOR [100]

From *Figure 10* we can observe that there are 9 DAISY descriptors. Each descriptor has a daisy flower shape, which is why it is named like this.



Each circle that formed the flower represents a region where the "+" sign depicts the pixel location where the computation of the convolved orientations maps center takes place, besides the radius of this region is proportional to the Gaussian kernels standard deviations.
Smooth transitions and rotational robustness degree can be caused by circles overlapping.
Considering the robustness in opposition to rotation, the radius of the outer regions is increased to be sampled equally than the rotational axis. [108]

FIGURE 11. DAISY DESCRIPTOR [108]

As it performs more computations than for example, a histogram, we think that this descriptor will take more time to obtain the desired values from the image but it could be more accurate.

3.3. Features Selection

Once we have analyzed the descriptors, that we consider that are going to be useful for the project development, we will continue with the study of a feature selector called PCA (Principal Component Analysis). We are uniquely going to analyze this selector because it is the most used in image processing and the most efficient.

3.3.1. PCA

The basic idea of PCA as an image feature selector is projecting the original data into a smaller dimensional space (smaller matrix of features). It selects from the original feature matrix, the most significant vectors (from highest to lowest importance) following a Singular Value Decomposition [171].

In image processing is interesting to preserves most of the variance of the image but reducing the dimensional space. The reduction is done dropping the lower singular values of the features vector.

In section 2.4 of this report, we have seen that there was a research made to analyze malware with images. In that case, the images analyzed were resized to 64x64 pixels gray-level pictures. The dimensionality of the data obtained is 4,096 values. However, the images of the same family look alike so the intrinsic dimensionality is much lower than 4,096 (for example, 250 values). The PCA features' selector can be applied to obtain a reduced set of values from the original vector but preserving the variance and the distinction between samples of images at the same time. In that way, the redundant information can be removed in the cases that the data is highly correlated.

The process of obtaining the reduced set of values has the following steps [114]:

1) <u>Obtaining the features matrix</u>: $f(x,y)$ is the function applied to the image that shows the different levels of gray color or the features for each pixel (x,y coordinates).

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & \dots & f(0,m-2) & f(0,m-1) \\ f(1,0) & f(1,1) & \dots & f(1,m-2) & f(1,m-1) \\ \dots & \dots & \dots & \dots & \dots \\ f(n-1,0) & f(n-1,1) & \dots & f(n-1,m-2) & f(n-1,m-1) \end{bmatrix} \quad (1)$$

2) <u>Adjust the matrix</u>: it is done by subtracting the features matrix with its mean to obtain columns with unitary variances and zero means.

$$adjusted_matrix = f(x,y) - \overline{f(x,y)} \quad (2)$$

3) <u>Covariance matrix:</u> the covariance of the adjusted matrix is calculated.
4) <u>Calculate eigenvalues and eigenvectors:</u> since the covariance matrix is squared, the eigenvalues and the corresponding eigenvectors are calculated.
5) <u>Obtain vector of eigenvectors (ve)</u>: from the covariance matrix, create a matrix of columns that will be the list of eigenvectors.

6) Final data: it is calculated by multiplying the transpose of the vector of eigenvectors with the transpose of the adjusted matrix of step 2.

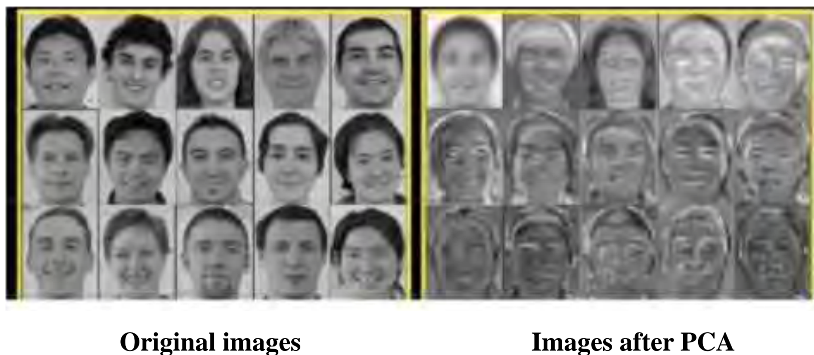
$$final_data = ve^{T} \times adjusted_matrix^{T} \quad (3)$$

7) Recover matrix of features: The original image can be obtained without compression with the following equation:

$$f(x,y) = ve^{T} \times final_data + \overline{f(x,y)}^{T} \quad (4)$$

Any components that represent a small variation in the data are going to be discarded, so the vector of eigenvectors is going to be reduced as the quantity of eigenvectors is less; consequently the *final_data* will have a smaller dimension. In this case, the image recovered using the equation (4) will be a compressed image with less detail than the original. This is the final purpose of this method.

This technique is often used in image face recognition, as it is explained in [109].



Original images **Images after PCA**

FIGURE 12. PCA FEATURES SELECTOR [167]

3.4. Machine Learning Classifiers

In image processing, an images classification process is performed in order to check the accuracy of the features extraction step and to extend its usage to face detection or medical systems. We will analyze the most used classifiers considering a supervised classification, as we know the class of each sample that will be classified. These classifiers are: KNN, NaiveBayes, DecisionTree and RandomForest.

3.4.1. KNN

KNN algorithm, also known as k-nearest neighbor algorithm, is the simplest machine learning classifier. The algorithm is based on classifying the testing objects taking into account the closest training samples in the space [113].

It consists in two steps: training and testing. In the training process it stores the features vectors and each label of the training objects, in this case the image family name or goodware/malware. In the classification process (testing), a majority voting process is taken to label the unknown sample equivalently than its k-nearest neighbors. Typically, the number of neighbors is 1, so the testing object is classified regarding the object nearest to it, meaning that it is considered that it belongs to the same class and it is classified in that way.

In the case that there are only two classes (goodware or malware), k must be an odd integer. Nevertheless, in multi-class classification can be a draw, even though the k number is odd. To solve this tie problem, it is commonly used the Euclidean distance, the most typical distance function:

$$d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \quad (1)$$

Being p and q two points in the Euclidean n-space and d the distance between these two points.

The following picture shows the whole KNN classification process:

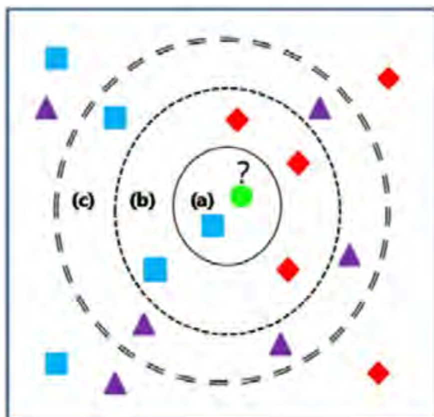


FIGURE 13. KNN CLASSIFICATION [113]

As we can see from *Figure 13,* the green circle (question mark) can be a square (a), a rhombus (b) or a triangle (c), depending on the value of K. If k=1, it will be a square; on the contrary if k=5, it will be a rhombus and in the case of k=10, it will be a triangle.

The main advantage of this algorithm is that it is really accurate if the classification is done with samples with different characteristics due to the fact the classification decision is taken in a neighborhood with similar objects (the objects with similar characteristics are close between them). On the other hand, the main drawback is that it considers the features equally, so that all the features have the same importance in the classification, causing errors when there is a small set of features.

In image processing, KNN is applied in face recognition like in [111] or in [112] to age estimation of a person.

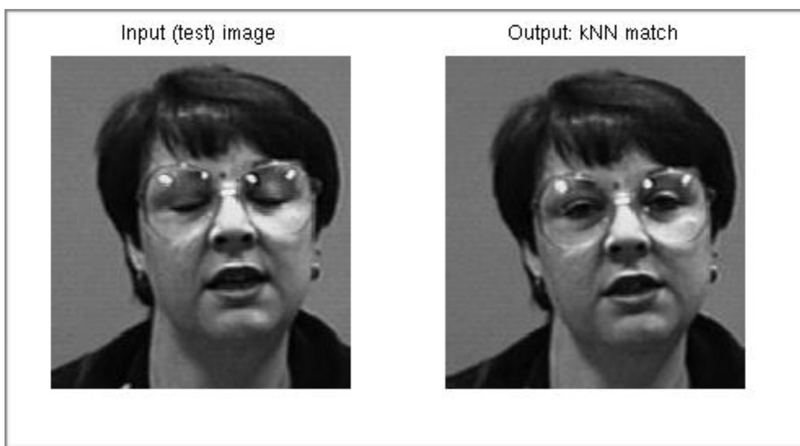


FIGURE 14. KNN FACE RECOGNITION [111]

FIGURE 15. KNN AGE ESTIMATION [112]

3.4.2. Naive Bayes

Naive Bayes is a simple algorithm that in some types of supervised learning classifications becomes a very efficient tool. Naive Bayes assumes that the value of an object feature is independent from the value of another feature of the same object, given the class label. For example, if a fruit is considered an orange when it is rounded and orange color, Naive Bayes does not consider the possible correlations between these features, it postulates that these two features independently participate to the probability that this fruit is an orange [173].

Accordingly, this algorithm is considered as a probability model [115] where a set of features $x = (x_1, \dots, x_n)$ are assigned a probability of contributing to k possible classes:

$$p(C_k \mid x_1, \dots, x_n) \quad (1)$$

However the above equation becomes unfeasible when the number of features is huge, so this formula is transformed using the Bayes theorem into the following:

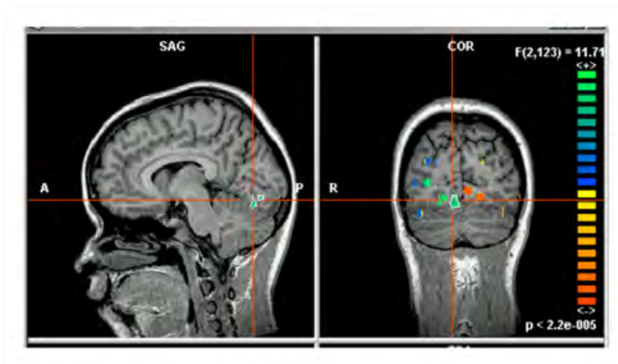
$$p(C_k \mid x) = \frac{p(C_k) \times p(x \mid C_k)}{p(x)} \quad (2)$$

The main advantage of this algorithm is that it demands a small quantity of training samples to do the estimation of the required parameters for the classification. Therefore, the time for training and testing is lower than other algorithms like KNN.

In image processing [116], the values are considered to be continuous because each value for x is a real-valued pixel. In this case, there is an assumption that states that these values follow a Gaussian (normal) distribution, so the likelihood of each feature value of the image is presupposed to be Gaussian:

$$p(x = v \mid c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}} \quad (3)$$

Being μ_c the mean and σ_c^2 the variance of the features value (x) of class c.



Some examples of the usage of Gaussian Naive Bayes is in medical images analysis [119] like in *Figure 16*.

FIGURE 16. NAIVE BAYES IMAGE PROCESSING [119]

3.4.3. Decision Trees

Decision Tree is a non-parametric supervised machine learning classifier. The classification process is based on learning decision rules formulated from the training features and predicting the value of an object using these rules. Usually, the inferred rules are if-else rules. Consequently, the deeper the tree is, the more complex the decision rules are going to be [117].

In the example of fruits of the previous section, the rules generated to define a fruit as an orange will be something like this:

```
IF color == orange AND shape == round:
          fruit = orange
ELSE:
          fruit = other fruit
```

The advantages of Decision Trees are [117]:

- Trees can be visualized (white box model) so they are easy to understand and interpret.
- They require little data preparing.
- They can handle multi-output problems.
- The computing cost is logarithmic.
- They work efficiently even though some assumptions are violated.

The drawbacks of Decision Trees are [117]:

- In some cases the problem of overfitting occurs. In these cases the created decision trees are so complex that the data generalization cannot be produced. Some solutions to overfitting are: pruning, setting a maximum depth for the tree or a minimum fixed number of leaves.
- There are rules that are difficult to be formulated in problems of multiplexer, parity or XOR.
- Decision Trees can be biased when one class predominates from the rest.
- Decision Trees can be wrong generated when there are small variations in the data.

Decision Trees are applied in image processing like in [120] and [121]. They outcome really effective in sonography medical analysis [118]:

FIGURE 17. DECISION TREES IN SONOGRAPHY ANALYSIS [118]

3.4.4. Random Forest

A Random forest is a meta estimator that follows a classification process based on dividing and averaging a number of decision trees into dataset's sub-samples [172].

This algorithm has the following steps [122]:

1) Select a number of bootstrap samples from the original dataset (n_{tree}).
2) For each bootstrap sample, build a unpruned tree with the following considerations (bagging process):
   2.1.) For each node, make a random sample (m_{try}) of the predictors instead of the choosing the best split.
   2.2.) Select the best separation from m_{try}.
3) Make a prediction with the new data by majority voting or by averaging of the predictions of the n_{tree} trees.
4) An estimation error rate from the training data can be obtained:
   4.1.) For each bootstrap iteration, predict the data using OOB (Out-of-Bag).
   4.2.) Add the OOB predictions (on average, each element of the data will be 36% times out-of-bag).
   4.3.) Calculate the error rate (OOB error).

The advantages of Random Forest classifier is [123]:

- This classifier works well with large datasets.
- It estimates the most important variables for the classification.
- It is not biased to any class of the dataset.
- It provides a balanced tree even the dataset is unbalanced.

In image processing, it is used in face recognition like in [124] or [125]:

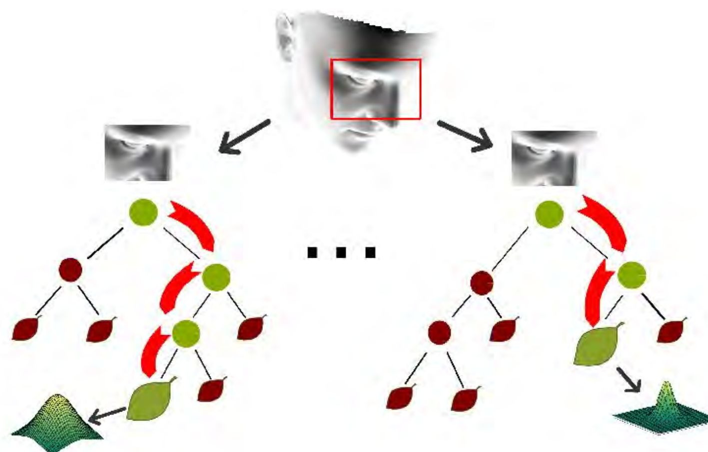


FIGURE 18. RANDOM FOREST FACE RECOGNITION [125]

3.5. Image Subtraction

When analyzing images, there is a technique that is based on subtracting one image from another [127] to detect the similarities between them or to separate the background from the image.

This process is done by subtracting pixel by pixel from the two images that are being analyzed. The pixel of *image1* will be subtracted with the corresponding pixel of *image2* (same position in the image matrix), so these two images must have the same size. The order of subtraction is not important because the result is taken in absolute value. This subtraction strategy is explained with the following example:

1) We have an initial digital image represented with a matrix. Each cell of the matrix represents a pixel and the value of each cell is the color of this pixel, between 0 (black) and 255 (white).

101	100	103	105	107	105	103	110
110	140	120	122	130	130	121	120
134	134	135	131	137	138	120	121
132	132	132	133	133	150	160	155
134	140	140	135	140	156	160	174
130	138	139	150	169	175	170	165
126	133	138	149	163	169	180	185
130	140	150	169	178	185	190	200

FIGURE 19. DIGITAL IMAGE

2) If we subtract the image with itself (in real life the image is different). We will have a zero values matrix. All the cells will be zero because if we start with the pixel(0,0) which has a value 101 and subtract it with itself, the result will be 0. The same thing happens to the rest of pixels, getting an image completely black.

101	100	103	105	107	105	103	110
110	140	120	122	130	130	121	120
134	134	135	131	137	138	120	121
132	132	132	133	133	150	160	155
134	140	140	135	140	156	160	174
130	138	139	150	169	175	170	165
126	133	138	149	163	169	180	185
130	140	150	169	178	185	190	200

-

101	100	103	105	107	105	103	110
110	140	120	122	130	130	121	120
134	134	135	131	137	138	120	121
132	132	132	133	133	150	160	155
134	140	140	135	140	156	160	174
130	138	139	150	169	175	170	165
126	133	138	149	163	169	180	185
130	140	150	169	178	185	190	200

=

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

FIGURE 20. IMAGE SUBTRACTION PROCESS

This strategy is useful in cases of superimposed images (photoshopped images) or motion detection. Referring to the later, the technique used is known as background subtraction (immobile objects will cancel each other whereas moving objects will be highlighted in the resulting image, after the subtraction of two images taken at slightly different times) [126].

Figure a

Figure b

Figure c

Figure d

FIGURE 21. IMAGE SUBTRACTION [170]

In *Figure 21* we can see different cases: *(a)* and *(b)* are pictures of the same scene but taken at different periods in time; *(c)* is the result of subtracting one image, either *a* or *b*, with itself (as the example explained before), it is a completely black image; *(d)* is the resulting image of subtracting *a* and *b* (the result is the same subtracting *a* minus *b* than *b* minus *a* because it is taken in absolute value), we can see than it detects that the scene has changed.

3.6. Requirements

This section contains the requirements that are going to be considered in the project designing. They are a negotiated agreement between the student and the tutor of the project and follow the standard IEEE 803. The requirements attributes are the following:

- **Title**: name of the requirement.
- **ID**: each requirement must have an identifier in order to ease its tracing in other sections.
  - Value: RXX (Requirement plus number). XX is a number which identifies uniquely each requirement.
- **Description**: it aggregates more information about the requirement in order to be better understood and explaining its function.
- **Priority**: The level of importance of each requirement.
  - Value: High / Medium / Low
- **Source**: the author of the requirement.
  - Value: Student / Tutor

The list of defined requirements is:

Title	Number of Samples
ID	R01
Description	The dataset must have at least two samples per class, in order to have at least one sample for training and the other for testing.
Priority	High
Source	Student

Table 3. Requirement 01 - Number of Samples

Title	Computing Time
ID	R02
Description	The total computing time of the whole process (image transformation and classification) for 100,000 samples must be less than 10 hours.
Priority	High
Source	Tutor

Table 4. Requirement 02 - Computing Time

Title	Accuracy
ID	R03
Description	The accuracy of the classification process must be higher than 50% (50% is considered as a random selection).
Priority	High
Source	Tutor and Student

Table 5. Requirement 03 - Accuracy

4. Design

Once we have performed a deep analysis about what to use in the experimental part of the project. We will define how to carry out the experimentation, adapting this design to the information obtained in the Analysis part.

In this section, we are going to adjust our datasets to fulfill the requirements specified; also we will propose the files that are going to be transformed into images. These images will be used in two experimentation lines to do the classification: the first one will be based on image subtraction and the second about extracting features of the image and then use one of the machine learning classifiers of the previous section. Besides, we will reason about the best way to implement these study lines.

4.1. Dataset Preparation

4.1.1. Dataset Adjustment

In the analysis phase we have studied each dataset deeply, however some of the classes contained in them could not fulfill the requirement *R01.* In that way, we are going to build two tables, one for each dataset, with its classes and the number of samples of each of them. In that way, the ones that do not satisfy the requirement will be discarded.

Initially, the two datasets are composed by:

Original M0droid Dataset

Class name	No. samples
Goodwares	200
Malwares	200

Table 6. Original M0droid Dataset

Original Malgenome Dataset

Class name	No. samples
ADR	22
AnserverBot	187
Asroot	8
BaseBridge	123
BeanBot	8
Bgserv	9
CoinPirate	1
CruseWin	2
DogWars	1
DroidCoupon	1
DroidDeluxe	1
DroidDream	16
DroidDreamLight	46

DroidKungFu1	34
DroidKungFu2	30
DroidKungFu3	309
DroidKungFu4	96
DroidKungFuSapp	3
DroidKungFuUpdate	1
EndOfDay	1
FakeNetflix	1
FlakePlayer	6
GamblerSMS	1
Geinimi	69
GGTracker	1
GingerMaster	4
GoldDream	47
Gone60	9
GPSSMSSpy	6
HippoSMS	4
JiFake	1
jSMSHider	16
KMin	52
LoveTrap	1
NickyBot	1
NickySpy	2
Pjapss	58
Plankton	11
RogueLemon	2
RogueSPPush	9
SMSReplicator	1
SndApps	10
Spitmo	1
TapSnake	2
WalkinWat	1
YZHC	22
zHash	11
Zitmo	1
Zsone	12

Table 7. Original Malgenome Dataset

As we can see, the M0droid Dataset satisfies perfectly the requirement *R01*. However, Malgenome Dataset has some families that do not fulfill this requirement, these families are: *CoinPirate*, *DogWars*, *DroidCoupon*, *DroidDeluxe*, *DroidKungFuUpdate*, *EndOfDay*, *FakeNetflix*, *GamblerSMS*, *GGTracker*, *JiFake*, *LoveTrap*, *NickyBot*, *SMSReplicator*, *Spitmo*, *WalkinWat* and *Zitmo*. All of them have only one sample so they cannot be used, in that sense, they are discarded in the experimentation and are not going to be taken into consideration.

Finally, the datasets with the corresponding families remain like this:

Adjusted M0droid Dataset

Class name	No. samples
Goodwares	200
Malwares	200

Table 8. Adjusted M0droid Dataset

Adjusted Malgenome Dataset

Class name	No. samples
ADR	22
AnserverBot	187
Asroot	8
BaseBridge	123
BeanBot	8
Bgserv	9
CruseWin	2
DroidDream	16
DroidDreamLight	46
DroidKungFu1	34
DroidKungFu2	30
DroidKungFu3	309
DroidKungFu4	96
DroidKungFuSapp	3
FlakePlayer	6
Geinimi	69
GingerMaster	4
GoldDream	47
Gone60	9
GPSSMSSpy	6
HippoSMS	4

jSMSHider	16
KMin	52
NickySpy	2
Pjapss	58
Plankton	11
RogueLemon	2
RogueSPPush	9
SndApps	10
TapSnake	2
YZHC	22
zHash	11
Zsone	12

Table 9. Adjusted Malgenome Dataset

4.1.2. Dataset Designing Decisions

Regarding the designing decisions, we have settled to use the Malgenome Dataset with testing purposes, that is, this dataset is going to be tested with several image descriptors, classifiers and the image subtraction technique. The reason of this decision is that inside this dataset there are more families which are similar between them (in the analysis part we have realized that some families have the same behavior so it derives unavoidably to similarities), so the classification process has to be more accurate, in order to being able to distinguish between one family and another. In that way, we will obtain the best classification strategy. Therefore, the M0droid Dataset is going to be used for corroborating the final classification resolution and testing if it is able to distinguish between goodware and malware, which is the final purpose of every antivirus and malware detector.

Besides, the Malgenome Dataset is unbalanced because there are families with a lot of samples and other ones with few samples, so the classification process becomes a difficult task meaning that it is preferable to use this as the testing dataset, confirming the previous reasoning.

4.2. Malware Transformation to Image

Once we have decided how to use the provided datasets and which families are going to be involved in this project, we are going to decide which files are going to be involved in the transformation phase to PNG images.

As already discussed in the Analysis section, inside every APK file there are the following files and folders: *AndroidManifest.xml*, *classes.dex*, *resources.arsc, res*, *assets* and *META-INF*.

We have decided firstly to follow an approach similar to static analysis, thus we are going to unpacked the APK file to obtain the previous mentioned directories and files. After unpacking all the application samples, we are going to select *classes.dex* to be converted into the desired PNG image. This decision is based on the idea that in *classes.dex* remains the most part of the code, so the main differences between applications are exhibited here. This way, these images are going to be used in the classification process and tested in order to choose the best classifier.

After deciding which is the best classification technique, we will try with the application itself with the chosen classifier. So that, each APK is not going to be unpacked, it will be transformed to an image directly. Then, the obtained images will be classified.

4.3. Classification Lines

After obtaining the images, we will process them considering two main strategies: image subtraction and features extraction with machine learning classification. These strategies will be tested measuring the time and the accuracy in order to know if they satisfy the specified requirements.

4.3.1. Classification with Image Subtraction

We propose an innovative technique for image classification. This technique is based on the idea that if we subtract one image with itself, we will obtain a completely black image (all the pixels will have value 0).

Considering this idea, we will develop a classifier that will decide to which family an unclassified malware belongs. To achieve this goal, the steps needed are:

1) Subtract each testing malware with all of the images of all the families of the training set.
2) Measure the color value of the resulting images.
3) The resulting image that has the lower color value, the one that is the nearest to black color (zero value), will be considered as the reference image.
4) Obtain the family name of the reference image by checking which family has been used in the subtraction process with this particular malware unclassified.
5) Once we know the family name, assign this name to the testing malware.
6) The malware has been classified to a specific class.

In this process they are going to be involved the training and testing set. They contain the samples of the adjusted datasets. We will perform some tests considering 60% of the samples of each family for training and the remaining 40% for testing; 80% for training and 40% for testing and finally with 1 image of each family in the training set and the rest in the testing set.

The accuracy of this classifier will be measured by simply checking the assigned family of the unknown malware and comparing it with the family that the unknown malware was taken from, before being put in the testing set. If they are equal, it means that the classification was right; otherwise it will not be accounted in the accuracy.

Finally, we will create a normalized confusion matrix to show graphically the results of the classification.

We think that this process will not be a time consuming activity because the subtraction operation is a really fast task. However, as it is an innovative approach we do not know if it will work as we expected.

Graphically, the classification process with image subtraction technique will be something like this:

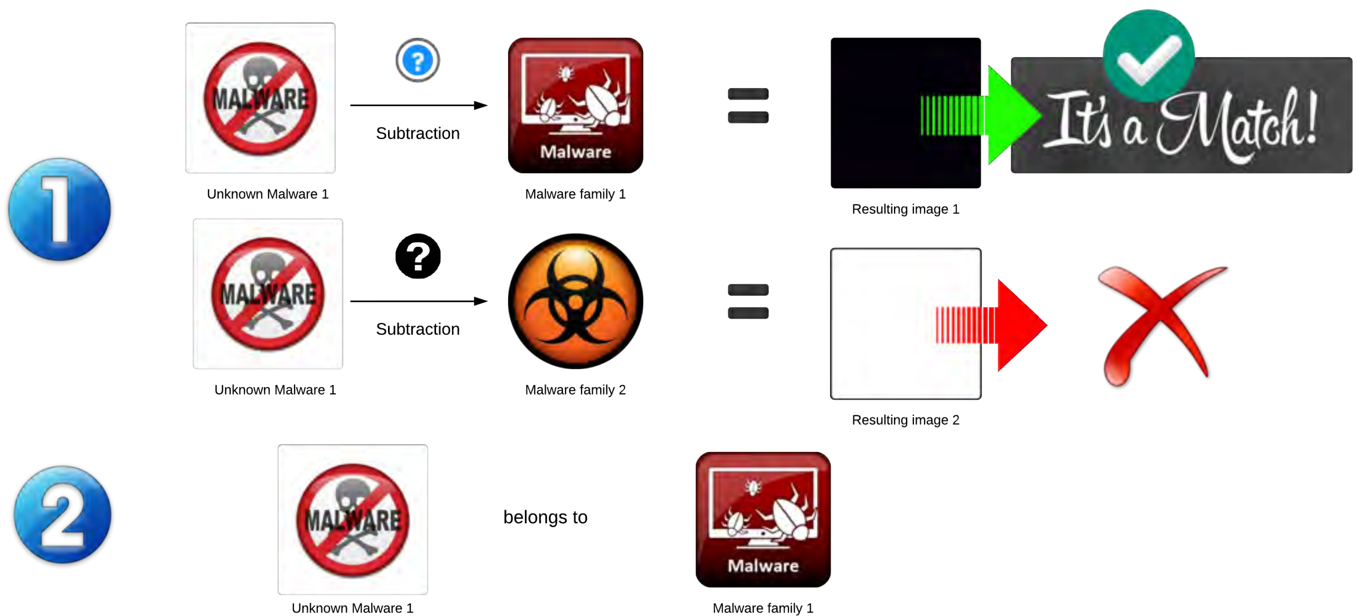


FIGURE 22. CLASSIFICATION WITH IMAGE SUBTRACTION

4.3.2. Features Extraction and Classification

The other approach is the one used in some malware detectors that are used nowadays. The idea is extracting some characteristics that identifies the specific malware and then perform a classification. In this case, we will extract features of the image that corresponds to a specific malware and use these features in the classification process.

In this approach we are going to use the image descriptors and the machine learning classifiers commented in the Analysis section:

- Image Descriptors: GIST, Histogram, Image to Graph and Daisy.
- Machine learning classifiers: KNN, Naive Bayes, Decision Tree and Random Forest.

The classification process has the following steps:

1) Extract the features of all the images of the adjusted dataset with one of the image descriptors.
2) Save the class of each image and associate it to the corresponding features extracted.
3) Perform a cross-validation process with one of the machine learning classifiers using the data obtained in the previous steps.

Once we have the classification results, we will create, as well, a normalized confusion matrix to show the results and we will measure the accuracy and the time, to check if they fulfill the requirements.

We are going to perform different tests reducing to the half each time the number of values extracted from the image descriptor. The number of features values analyzed will be:

- 960 values: they are the maximum number of values that GIST descriptor can retrieve, so we decided to start from this with the rest of descriptors in order to perform a good comparison between classifiers.
- 400 values: the next number of features values will be near the half of 960.
- 200 values: we reduce by half the previous number.
- 100 values: halving the previous number.
- 50 values: the last reduction of the tests performed.

Additionally, after analyzing all the results with each classifier and image descriptor, we will perform another test to reduce even more the time, which is using PCA features selector to obtain the best features of an image and increase the accuracy of the final classifier.

Finally, with all the experiments done we will choose the best classifier and perform a final conclusion.

Graphically, this approach is represented in the following way:

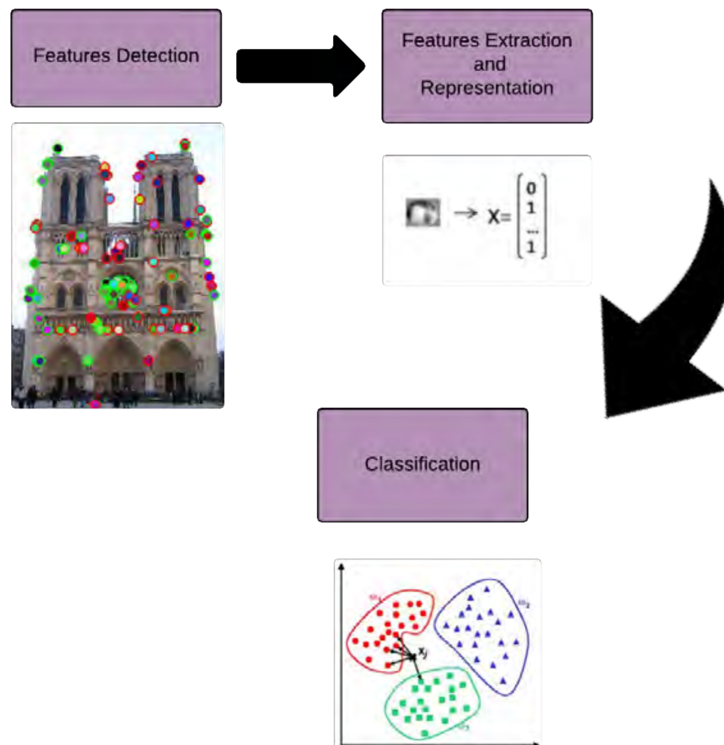


FIGURE 23. FEATURES EXTRACTION AND CLASSIFICATION

4.4. Programming Language Determination

Once we have the most important design decisions chosen, we are going to explain in this section the main reasons why we have determined to implement this project with Python.

4.4.1. Why Python?

Some of the characteristics that motivate us to choose Python as the programming language used for this project are the followings [129]:

- Simplicity: Python is considered as a minimalistic and simple programming language. It is easy to read and to understand, so it is similar to write in pseudo-code, which allows the developer to concentrate mostly in solving the problem than in the programming language itself.

- Open Source and Free: it is a FLOSS (Free Library and Open Source Software) example that is based on sharing knowledge. It can be distributed in copies and being affected by changes and improvements.

- Object Oriented: it supports object-oriented and procedure-oriented programming. The former allows writing a program based on objects that combine data and functionality. With the later a program is built using functions that are reused pieces of other programs.

- High-Level Language: the programmer does not have to consider low-level programming issues such as dealing with the memory.

- Extensible: it can add other programs written in C/C++.

- Embeddable: it can be included inside C/C++ programs to offer "scripting" capabilities to the users of a program.

- Portable: it works in many platforms due to its open-source nature. A program in Python can be executed in different platforms with any change at all. It can be run in *Windows, Linux, Mac OS X, Solaris, FreeBSD, OS/2, AROS, AS/400, Amiga, BeOS, z/OS, OS/390, Palm OS, PSion, QNX, VMS, Acorn RISC OS, PlayStation, VxWorks, Windows CE, Sharp Zaurus* and *PocketPC*.

- Interpreted: it is the main difference between other languages like C or C++. In these languages the program is converted from this source language into binary code by the compiler. When the user runs the program, it is copied from the hard disk and is run thanks to the loader/linker. On the contrary, in Python there is no need for compiling the code to binary. The program is run directly from the source code, Python is in charge of converting this source code into an intermediary form (bytecodes) and then into computer language. After this process, Python runs the program. All of them increase the Python simplicity and portability due to the fact that the program is not compiled, so if it is copied to another computer it will work.

- Extensive Libraries: it follows a "Batteries Included" philosophy, in which the standard library is really huge allowing the development of different kind of programs of several knowledge branches like cryptography, databases, web browsers, etc. Additionally, it allows adding new libraries. Some external libraries like GIST library will be used in this project.

Considering all of these features and as in this project the most important thing is obtaining an accurate classification instead of how it has been coded, we conclude that Python is the suitable programming language for the project implementation part.

4.4.2. Comparison with Another Languages

In order to emphasize our choice, we are going to compare Python with other common programming languages [130]:

- Java: The running time for Python programs generally is higher than Java programs but on the other hand, the development time is lower (between 3 and 5 times shorter). There are not variables or types of arguments declarations, the dictionary types and polymorphic lists are very powerful. Java and Python can make a good combination because Java is more a low-level implementation language and Python is considered as a "glue-language". Due to this fact, components can be developed

in Java and then being combined to be part of Python applications. Also, some programs can be firstly being written in Python as a prototype and then being implemented in Java. There is a project development based on this idea that tries to translate Python source code to Java byte code.

- JavaScript: Python object-oriented is similar to JavaScript. Although both of them use simple functions and variables and have a similar programming style, in Python there is the possibility of the existence of classes and inheritance for larger programs and code reusing.

- C++: Between Python and C++, some of the differences are similar than the ones of Java. In the case of the length of the code, for C++ it is 5-10 times larger than in Python and also it is considered as a “glue-language” to combine C++ components.

As a conclusion, we can say that our choice is based on the idea that in order to do a deep analysis and obtain the best classification technique we have chosen Python. It is a prototype so we are not going to focus in coding, we do not want to invest so much time in this phase. Although Java or C++ are faster languages in terms of performance, we prefer to do more tests with different classifications to get the best one, instead of trying to solve errors in coding as these programs are more difficult in terms of coding than Python. Once the best classification process is selected, it can be implemented in another language but we are only going to propose a prototype that can be developed later.

5. Implementation

Once we have designed how this project is going to be implemented, we continue with the implementation decisions taken. In that way, the following section has all the determinations regarding implementation issues and the problems found during this phase of the project. We are going to start with the description of the project environment, then, we mention the libraries used, the steps needed to obtain the classification results and the problems found during the whole phase. In the description of the classification steps we will show how the accuracy and the time are measured.

5.1. Project Environment

This project has been developed using a VMWare Virtual Machine with Ubuntu 15.10 64-bit as a guess Operating System in an iMac 2011. The CPU is an Intel® Core™ i5-2400S 2.50GHz.

We have decided to use a virtual machine because we are dealing with malware so it is preferable to work in a safe and isolated environment to avoid affecting the host Operating System (Mac OS X El Capitan 10.11.4). We have chosen Ubuntu 15.10 as guess OS because it is the most common Linux free distribution, we have not used the last version of Ubuntu (16.10) because the GIST library is not supported in this last version and we have faced some errors with it.

5.2. Libraries Used

As it is explained in the design section, one of the advantages of Python is the possibility to include external libraries to the program that is being implemented. In that way, we have chosen several libraries to implement this project. For the installation of these libraries we need `pip` (standard Python installation program) to be installed in our system. It is usually included in the last versions of Python although we have to upgraded it in order to get it working properly.

- Pandas confusion matrix (pandas_confusion 0.0.6). *Pandas* is a BSD-Licensed open-source high performance library which provides structures and tools for data analysis [131]. With this we will build the final confusion matrix with the results of the classification. It is used in the both implementation lines (subtraction strategy and features extraction and classification).
  To install *pandas_confusion* we have to run the following command in the terminal shell: `pip install pandas_confusion`

- Python Image Library (PIL): it adds powerful image capabilities to Python. It is used in both study lines. To install PIL, the command needed is: `pip install Pillow`

- Numpy: this package is included in Ubuntu so we do not need to install it. It contains powerful scientific capabilities such as array objects, linear algebra functions, etc. It is used in both study lines.

- Matplotlib: it is already provided with Python. We have used this package to show the confusion matrix previously computed.

- Leargist (pyleargist 2.0.5): it is the library for GIST image descriptor used in the second study line. Firstly, we must have the following libraries installed: *libfftw3* with development headers, *gcc*, *PIL* and *numpy*. We have to download the *leargist* package from https://pypi.python.org/pypi/pyleargist and, after extracting the contents, type these two commands in the terminal: `python setup.py build` and `sudo python setup.py install` (with the later we need root privileges).

- Scikit-learn: it is an open-source library (BSD License) that contains machine learning classifiers for Python among other tools for data mining and analysis. To install it, we need to type this command in the terminal: `pip install -U scikit-learn`. This library is used in the second study line, in the features extraction part as well as in the machine learning classification.

These are the main external libraries used in this project, although some of them are already included in current Python versions or in Ubuntu. Another packages used in the project implementation, which do not need to be installed, are: *pickle, sys, time, array, os, glob*, among others. The code can be seen in Annex I.

5.3. Project Implementation Steps

After the installation of the necessary libraries, we can proceed with the project implementation. As it is explained in the design phase, there are going to be two study lines: one for image subtraction classification and another for features extraction and classification. These two processes have two common previous steps: obtaining the classes.dex files from the dataset and converting them into images. All of this procedure is explained in this section. In case we use directly the APK file (packed application), the step of obtaining the classes.dex is not considered and the transformation of the image is done with the application itself. Each subdivision of this section has been implemented with a different Python script that can be found in the Annex I.

5.3.1. Unpacking APK Files

As it is explained in the previous phase, we are going to start with the analysis of the *classes.dex* file. It is included inside each APK. Firstly, we have to have prepared the datasets by removing the families that are not considered. We have done it manually as there are not too many families that we have to discard.

Once we have the datasets adjusted, the M0droid dataset will have two folders (goodware and malware) and Malgenome dataset 33 directories (33 families). Inside the folders of M0droid dataset there are files with the extension *.bening* and *.malware*, corresponding to goodware and malware samples respectively. Although the usual Android applications have the *.apk* extension, they have the same files inside and are processed in the same way that an *.apk* file. Inside the folders of Malgenome dataset there are *.apk* files for each family.

As there are two datasets that include different file extensions, we have created two scripts *extract_F.py* and *extract_GM.py* (Annex I), for Malgenome and M0droid datasets respectively, to obtain the *classes.dex* file.

This process has the following steps:

1) Obtaining the file path from the dataset introduced as an argument by the user with the following command: `python extract_X.py <dataset name>`.

2) A folder called "*families*" or "*good_mal*", depending on the dataset introduced, is created, to extract the files inside, in the location where the previous command was typed. This folder will contain the different families and inside each family it will be a folder for each sample, which has the same name than the original sample, with the data extracted (*classes.dex* among other files, as commented in the analysis section).

3) There is a loop to create the different folders for each sample of each family and to extract the information contained inside the samples, considering the extension issue previously commented. This is done with a zip extract function. In case of the M0droid files extraction, there are some files that are encrypted, so these samples are discarded because we cannot access to them as we do not have the password. This is the main reason why we have created two different scripts for the extraction of the files.

4) After this extraction, we have the datasets with the samples unpackaged. In the image conversion phase, we will choose only the *classes.dex* contained inside these folders.

As a summary, this process is based on going through different folders and extracting the information contained in them, considering the files extension and the possibility that some of the files are encrypted. This information is stored in other directories previously created in a specific destination. This unpacking process must not be executed in case of transforming an APK file directly to an image.

5.3.2. Image Conversion

After obtaining all the APK files unpacked, we are going to transform the classes.dex into images, the rest of files are not going to be considered. The conversion process to PNG images is performed in the file *convert2image.py* (Annex I):

1) Obtain the main directory (*families* or *good_mal*) location to be processed. This directory has been created before and contains all the files included inside each APK file. It is introduced as an argument when executing the command:
```
python convert2image.py <directory with all the apks unpacked>
```

2) Then, we go through each family inside this main directory and select the *classes.dex* file. In some cases of the M0droid dataset, there were malware samples that do not have this file, so an error message is printed in the terminal ("*classes.dex does not exist*").

3) As we see in previous sections, the width of an image depends on the file size that is going to be transformed. We use *Table 2* as a reference. Once, we get the classes.dex file, we check its size (in bytes) and depending on it, we assign a value for the width of the final image.

4) We create an array of unsigned integers of 1 byte (uint8) data types in which we will copy the contents of the *classes.dex*. This is due to the fact that an image will have values for each pixel between 0 and 255, so the proper data type is uint8.

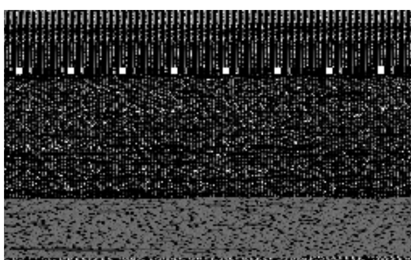
5) Once we have the *classes.dex* contents copied, we reshape the array to have the dimensions of the final image: (height x width). As we have said, the width is fixed depending on the file size and its height is calculated considering the mathematical equation of the area of a square, being the $\textbf{\textit{Area}}$ the file size:

$$\text{Area} = \text{width} \times \text{height} \implies \text{height} = \frac{\text{Area}}{\text{width}} \quad (1)$$

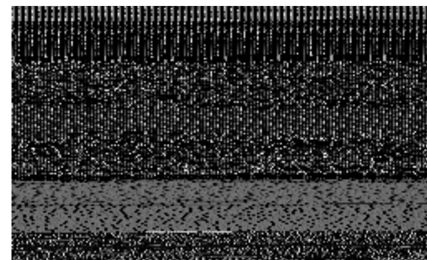
6) The image is built as a *numpy* array of 256 unsigned integers (digital image matrix). We use the function *scipy.misc.imsave(file_name, array)* to convert this array into an PNG image with the same name than the original APK file. It is saved inside the corresponding family folder of the main directory (*families* or *good_mal*) where the original APK file belongs to.

As a summary, this process is based on acquiring the contents of the *classes.dex* file and arranging this data with a fixed width, which is assigned depending on the file size, and with a height calculated from this width and file size. The matrix obtained is transformed into a PNG image.
This process will be extrapolated later to transform directly a packed application to an image. The only difference between the conversion of an unpacked or packed application is that instead of using the contents of the *classes.dex* file (the app has to be unpacked before), we are going to transform the *.apk* file directly without being unpacked.

Here we can see two examples of images, from the M0droid dataset, obtained after this conversion process. The image on the left-side corresponds to the *classes.dex* of a goodware file and the right-side image is the *classes.dex* of a malware application. We can appreciate some differences between the two images that will be used to obtain a classification pattern. However, there are parts of both images that are very similar, meaning that the differences between goodware and malware are presented in few bytes.



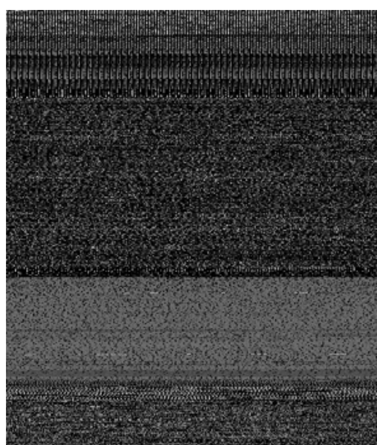
Goodware image



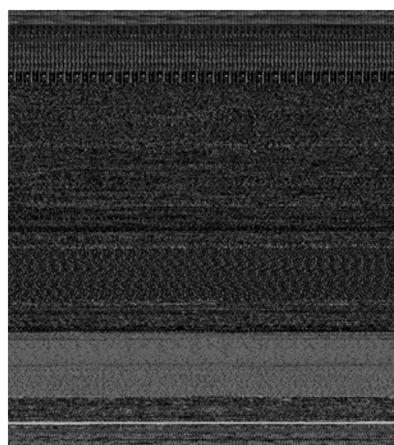
Malware image

FIGURE 24. GOODWARE AND MALWARE IMAGE EXAMPLES (M0DROID)

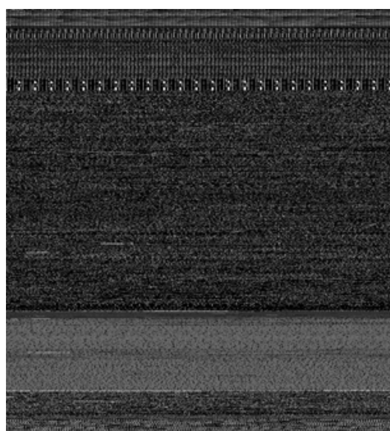
After the conversion process, four examples from Malgenome dataset are the ones above (*Figure 25*). We can see that images from *KMin* and *RogueLemon* families are similar, although they come from different families and have different behavior, showing the reutilization of software to create new malware. Regarding the images of DroidKungFu3 and DroidKungFu4, we can see that they have similarities, but *DroidKungFu3* is darker than *DroidKungFu4* (*DroidKungFu3* has more quantity of black color) meaning that it has less information than *DroidKungFu4*, this is reasonable because DroidKungFu4 is an improvement of DroidKungFu3, so it contains more information than its previous version.



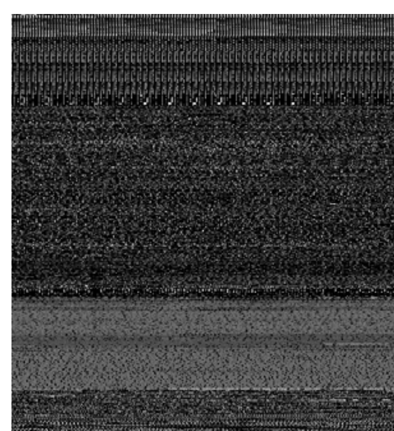
KMin image



RogueLemon image



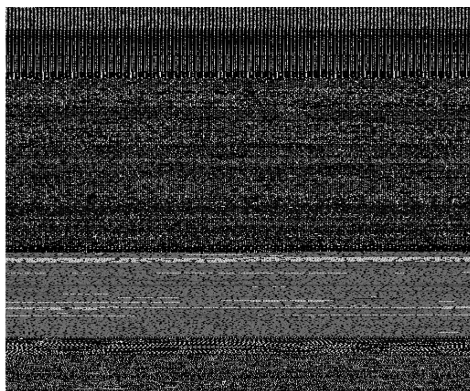
DroidKungFu3 image



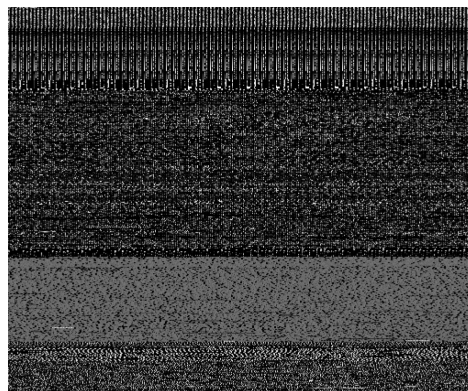
DroidKungFu4 image

FIGURE 25. DIFFERENT FAMILIES IMAGE EXAMPLES (MALGENOME)

In *Figure 26* we can appreciate that there are two samples of the ADR family, they look really similar as they have the same characteristics.



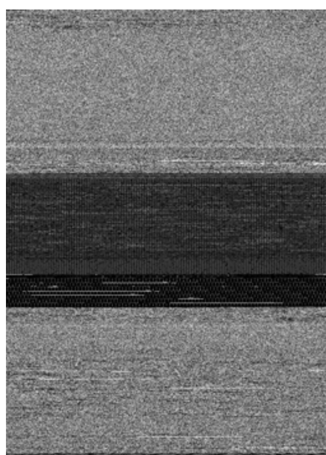
ADR example1 image



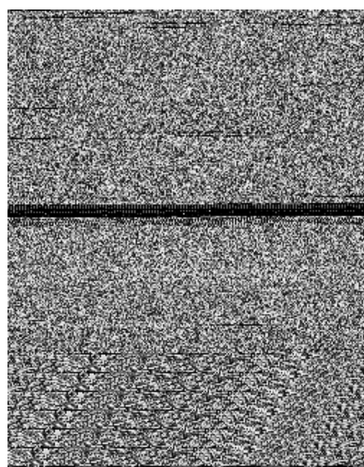
ADR example2 image

FIGURE 26. ADR EXAMPLES (MALGENOME)

A packed application will look like in *Figure 27*. The one on the right is a goodware application, and the one on the left is a malware APK file. We can observe that the malware images are lighter because they contain more information.



Goodware APK packed



Malware APK packed

FIGURE 27. PACKED APPLICATIONS IMAGES

5.3.3. Subtraction Classification

Our first approach is the implementation of a new classification technique based on image subtraction (*sub_classif.py* - Annex I), as it is explained in the design section.

This classification process is performed considering the following steps:

1) Shuffle randomly the samples of each family to avoid obtaining the same images for testing and training each time the program is run (analysis of results purposes).

2) For each family, calculate the number of images destined for training and testing sets and gather them in the corresponding set.

3) Select one image of the testing set and obtain its width and height. This step and the followings will be done for each image in the testing set with a for-loop.

4) In a for-loop, we open, in each iteration, one of the images of the training set obtaining its size (width and height).

    4.1.) Check the width and height of the testing and training images and resize them in case they are not equal, considering the lowest values for the width and height.

    4.2.) Once we have both images with the same size, we subtract them using *ImageChops.difference(image2,image1)* function. This function is inside the *PIL* library.

    4.3.) Calculate the color value (between 0 and 255) of the resulting image by adding the color value of each pixel of the image.

        4.3.1) In case that the resulting color is zero, meaning that both images are identical, we stop this process and classify the testing image to the family which the current training image belongs to. Go to step (7).

        4.3.2) If the resulting color is not zero, it is saved in an array that will contained all the color values of the subtraction process for a specific family.

    4.4) From the array of all the color values of a family, choose the minimum value and store it in an array that contains the model color value of the subtraction process of a family with the current testing image.

5) Once we have performed the subtraction between testing image with all the training images of all the families, we choose the minimum value of the array that contains the model values of all the families involved in this process.

6) As this model value belongs to the color of the resulting image obtained with the subtraction between the current testing image and a training image of a family, we found the family name involved in this specific operation and label the testing image with this name, in other words, we classify the testing image to the family of the training image involved in the operation that results with an image that has the lowest quantity of color (the most similar to black color).

7) To check if the classification process was right, we check if the index of the predicted family is the same than the index of the real family which the testing image belongs to.

8) Save the predicted family label in an array of predicted classes and the actual family name in another array with the real classes.

9) Once we have performed this process for all the testing images, we compute the confusion matrix with the two arrays that contains the predicted and actual family names. To do this, we call the function *ConfusionMatrix(y_actual, y_predicted)* from the *pandas_confusion* library. Besides we use the function *print_stats()* to obtain statistics from the classification process such as the accuracy or f1-measure.

10) Finally, we show graphically the resulting confusion matrix normalized with the *matplotlib* library. We normalize the confusion matrix because the number of samples for each family is not the same, so in this way we obtain a weighted value for each family.

In addition, apart from the accuracy, we are going to measure the time needed to run this program. To do this we will use the command:

```
time python sub_classif.py <directory with images to be classified>
```

With this command, they will be printed in the terminal three values:

- REAL: real elapsed time used by the executed process (seconds).
- USER: number of seconds used by the CPU in user mode.
- SYSTEM: number of seconds used by the CPU in kernel mode.

We are going to consider only the REAL value, as it represents the total elapsed time of the whole process.

If we do not want to measure the time, we just remove the word “time” of the previous command:

```
python sub_classif.py <directory with images to be classified)
```

One example of the resulting confusion matrix obtained after this classification is the one above. For this test we consider 60% of samples for training and 40% for testing. All the confusion matrices corresponding to the different tests performed can be found in Annex II.

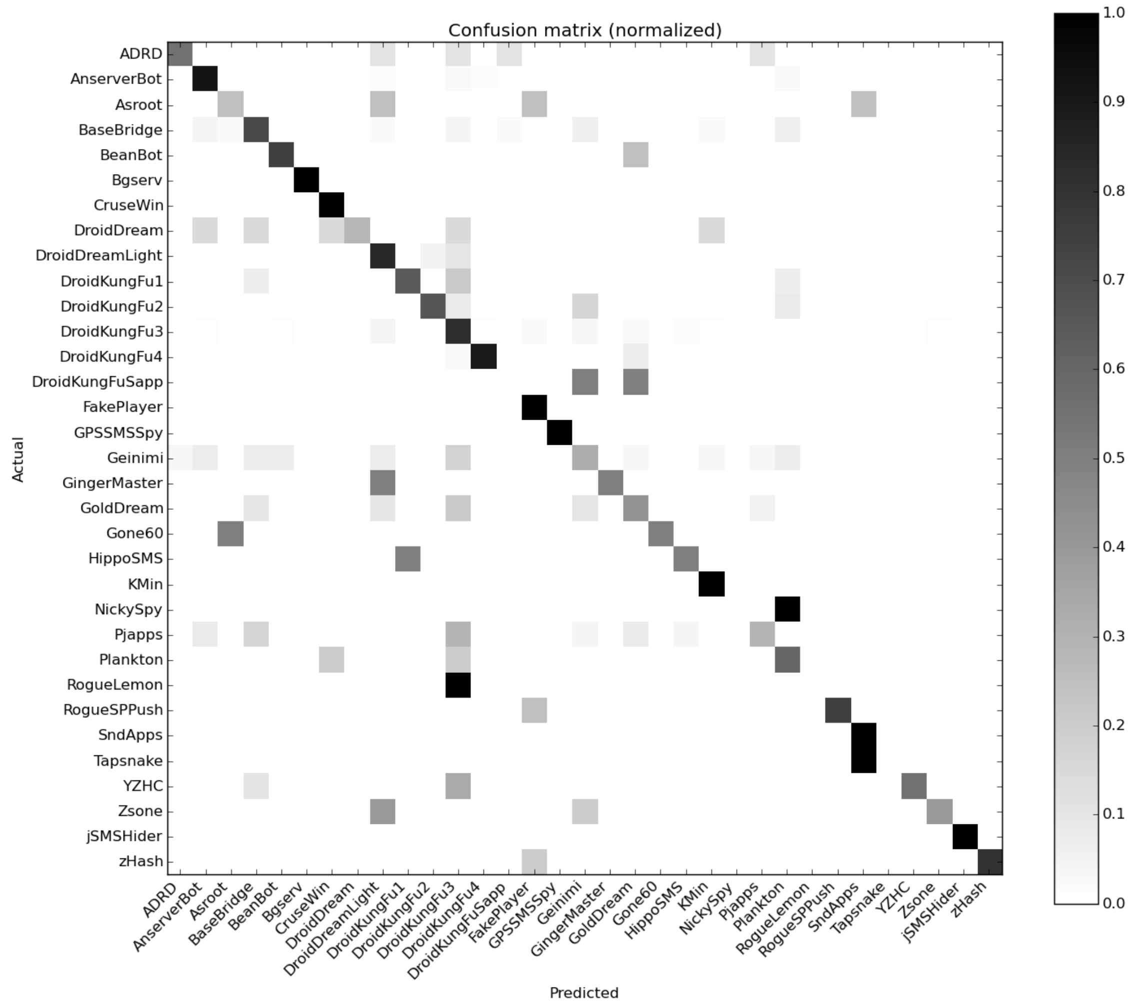


FIGURE 28. CONFUSION MATRIX SUBTRACTION CLASSIFICATION

5.3.4. Features Extraction and Classification

Our second approach is based on continuing with the design implemented in other researches. We are going to extract characteristics of the images and then use these features to obtain a pattern with a classifier to distinguish the samples into different classes.

These two parts are gathered in a script called *classif.py* (Annex I). To run this script we just need writing in the terminal the command:

```
python classif.py <directory with images to be classified>
```

We are going to perform different tests as well, so for time measurement we have to add the word "time" in the previous command and we will obtain a measure for the real, user and system time, as it is explained in the previous classification strategy :

```
time python classif.py <directory with images to be classified>
```

In this procedure they are going to be involved to important matrices: *X* and *y*. The former contains all the features extracted for each image and the latter contains the family name (label class) of these images in the same order they are stored in *X* matrix. Therefore, the first position of the *X* matrix contains the features extracted of the first image and in the first position of *y*, the label of this first image; the same happens for the rest of the images. This procedure has the following steps:

1) Obtain the number of samples of each family/class and store the label of each image (the family name which it belongs to) in the *y* matrix.

2) Start the process of extracting the features of all the images with a specific descriptor.
With tests purposes, we are going to measure the time it takes to get all the features values, so we start measuring the time using *time.time(),* which will retrieve a floating number representing the seconds since the epoch. When this extracting process ends, we will measure again the time and we calculate the time passed by subtracting the final value with the initial value of time.
Besides, we are going to change the number of descriptors retrieved, starting from 960 until 50, as it is commented in the design phase.
For faster computation we have decided to resize all the images to 35x35 size, this is due to the fact that daisy descriptor cannot analyze bigger images, so in order to compare properly all the descriptors we have established these values for the width and the height. Moreover, we have checked that if we increase these values, the accuracy remains the same and the time increases.
The descriptors used are the ones studied in the analysis phase, the values retrieved are going to be copied to the X matrix, considering the number of descriptors specified (from 960 to 50):

   - GIST: we have used the leargist library to obtain the descriptors with the function *leargist.color_gist(image).*

- Histogram: the *numpy* library of Python has a histogram computation function *numpy.histogram(numpy.ravel(image),no_des,[0,no_des])*. However, the image introduced to this function must be a flattened continuos array, for this purpose we use the function *numpy.ravel(image)*.

- Daisy: for this image descriptor we have used the scikit-learn library (*daisy(image)*). This descriptor retrieves two vectors of size 200 each, we have used *numpy.hstack(array1, array2)* to put consecutively these 200 values to form a row of 400 values that defines the total number of features values retrieved, which are going to be stored in *X* matrix for each image. As we commented before, we are going to start with 960 values retrieved but in this case we make an exception and we start from 400, as it is the maximum possible values retrieved.

- img_to_graph: for this image descriptor we have also used the *scikit-learn* library (*img_to_graph(image)*). We will obtain the features values from the graph retrieved.

Additionally, we have performed tests using two descriptors at the same time: histogram with daisy and GIST with daisy, to check if the accuracy improves. In this case we also use *numpy.hstack(array1, array2)* to gather both descriptors into *X* matrix.

3) Once we have the features extracted and the labels of all of the images, we continue with the supervised classification process, all the functions used in this step are part of the scikit-learn library.

3.1.) Divide the data into k-folds for grouping it into testing and training sets. In this process we do a random shuffle as well. We use *StratifiedKFold(y,kfold)* as a cross-validation iterator because it provides train and testing indices returning folds that preserves the percentage of samples per class. We choose k=2 (2 folds) because there are classes with only 2 samples.

3.2.) Select the supervised classifiers with 2-fold cross-validation. The classification has been made with one of the classifiers studied in the analysis phase:

- KNN: with *KNeighborsClassifier(no_neighbors,[weights])* we have establish 1 neighbor because it is the usual value it gets. Although this function uses by default uniform weights (the final value of a query point is computed by nearest neighbor voting), in this circumstance its better than the nearest neighbors contribute more to the fit than the others, so they must have a bigger weight than the others. To do this, we have decided to use the option *weights='distance*' (instead of the default one *weights='uniform*') because in that way the distance weights from query point to its neighbors are assigned inversely proportional (more distance means less weight).

- Naive Bayes: as it is commented in the analysis phase, it is preferable to use Gaussian Naive Bayes for image processing so we have used *GaussianNB()* function to perform the classification.

- Decision Tree: we have used the classification function *DecisionTreeClassifier(random_state)* with a zero value for the seed used by the random number generator, as it is the usual value.

- Random Forest: the random forest classification is performed with the function *RandomForestClassifier(n_estimators)* with 10 estimators as it is the value by default.

3.3.) Select the most relevant features with PCA to reduce the training and testing time. We have decided to use 25 components after performing several tests (*PCA(n_components)*).

3.4.) Perform the training with *fit(X_train_pca, y_train_pca)* and testing with *predict(X_test_pca)*. In these two processes we have measured the time too, for a future comparison between classifiers. The predicted values are stored in an array of predicted classes and the actual family name in another array with the real classes.

4) After the classification process, we compute the confusion matrix with the two arrays that contains the predicted and actual family names. To do this, we call the function *ConfusionMatrix(y_actual, y_predicted)* from the *pandas_confusion* library. Besides we use the function *print_stats()* to obtain statistics from the classification process such as the accuracy or f1-measure.

5) Finally, we show graphically the resulting confusion matrix normalized with the *matplotlib* library. We normalize the confusion matrix because the number of samples for each family is not the same, so in this way we obtain a weighted value for each family.

One example of the resulting confusion matrix obtained after this classification is the one above. All the confusion matrices corresponding to the different tests performed can be found in Annex II.

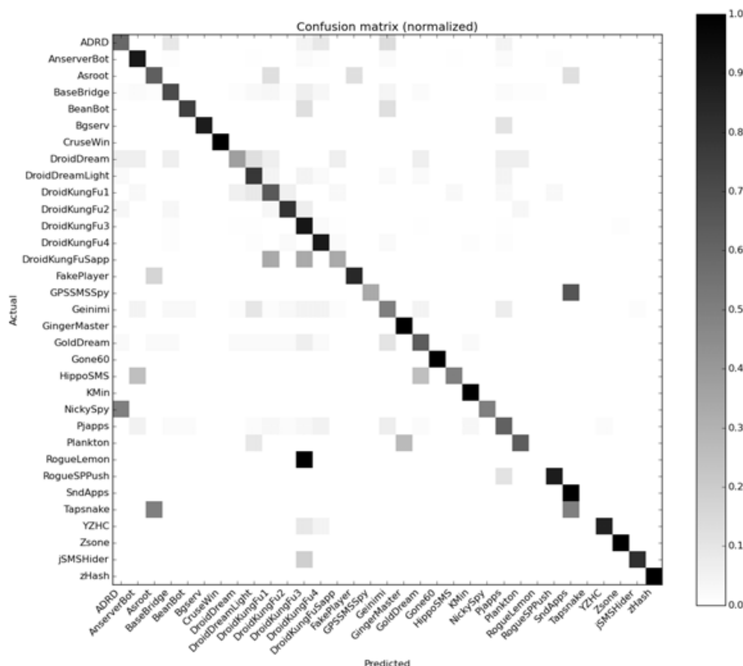


FIGURE 29. CONFUSION MATRIX FEATURES EXTRACTION + CLASSIFICATION

5.4. Problems Found

Regarding the problems found in the implementation phase, we have to mention the following obstacles:

- The impossibility to unpack and transform into images some samples of M0droid dataset because we were not able to decrypt these samples as we did not know the password or they do not contain the *classes.dex* file. Therefore, these samples were discarded, remaining 200 goodware samples and 197 malware samples for the classification. The problem is solved by transforming the application directly to an image, in that way, we do not have to access to each file inside the application.

- Daisy descriptor can retrieve at maximum 400 values, so we cannot test this descriptor with 960 values as we have planned initially.

- The *pandas_confusion* is a work in progress, so some statistics are not calculated. However, as we are only considering the accuracy, we are not affected by this problem.

- The *leargist* library does not work in Ubuntu 16.10, so we have to use a previous version 15.10.

6. Performance Evaluation

6.1. Tests Description

As we have explained in the design section, we have planned to use the Malgenome dataset to perform the different tests. In these tests we have measured the accuracy, the time and we have analyzed the confusion matrices obtained. Due to the fact that there are two common steps in the project development, unpacking the app and transforming it to an image, we have divided these phase in two parts, considering the two classification strategies:

- Subtraction classification: We have performed tests changing the size of the training and testing sets. We have forced that the training set has a specific percentage of images of each family and the testing set has the remaining images. In the following table, we summarize the different samples assigned for each set of the different tests:

	No. training images of each family	No. testing images of each family
TEST A	80 % of the images	20 % of the images
TEST B	60 % of the images	40 % of the images
TEST C	1 image	All the images of the family - 1

Table 10. Subtraction Classification Tests

- Features extraction and classification: These tests are based on changing the image descriptors, the number of descriptors retrieved and the machine learning classifiers. The following table summarizes all the tests performed:

Table 11. Feature Extraction + Classification Tests

	Image descriptor[no.descriptors]	Machine learning classifier
TEST 1	GIST[960]	KNN
TEST 2	GIST[400]	KNN
TEST 3	GIST[200]	KNN
TEST 4	GIST[100]	KNN
TEST 5	GIST[50]	KNN
TEST 6	GIST[960]	Gaussian Naive Bayes
TEST 7	GIST[400]	Gaussian Naive Bayes
TEST 8	GIST[200]	Gaussian Naive Bayes
TEST 9	GIST[100]	Gaussian Naive Bayes
TEST 10	GIST[50]	Gaussian Naive Bayes
TEST 11	GIST[960]	Decision Tree
TEST 12	GIST[400]	Decision Tree

TEST 13	GIST[200]	Decision Tree
TEST 14	GIST[100]	Decision Tree
TEST 15	GIST[50]	Decision Tree
TEST 16	GIST[960]	Random Forest
TEST 17	GIST[400]	Random Forest
TEST 18	GIST[200]	Random Forest
TEST 19	GIST[100]	Random Forest
TEST 20	GIST[50]	Random Forest
TEST 21	Histogram[960]	KNN
TEST 22	Histogram[400]	KNN
TEST 23	Histogram[200]	KNN
TEST 24	Histogram[100]	KNN
TEST 25	Histogram[50]	KNN
TEST 26	Histogram[960]	Gaussian Naive Bayes
TEST 27	Histogram[400]	Gaussian Naive Bayes
TEST 28	Histogram[200]	Gaussian Naive Bayes
TEST 29	Histogram[100]	Gaussian Naive Bayes
TEST 30	Histogram[50]	Gaussian Naive Bayes
TEST 31	Histogram[960]	Decision Tree
TEST 32	Histogram[400]	Decision Tree
TEST 33	Histogram[200]	Decision Tree
TEST 34	Histogram[100]	Decision Tree
TEST 35	Histogram[50]	Decision Tree
TEST 36	Histogram[960]	Random Forest
TEST 37	Histogram[400]	Random Forest
TEST 38	Histogram[200]	Random Forest
TEST 39	Histogram[100]	Random Forest
TEST 40	Histogram[50]	Random Forest
TEST 41	Image To Graph[960]	KNN
TEST 42	Image to Graph[400]	KNN
TEST 43	Image To Graph[200]	KNN
TEST 44	Image To Graph[100]	KNN
TEST 45	Image To Graph[50]	KNN

TEST 46	Image To Graph[960]	Gaussian Naive Bayes
TEST 47	Image To Graph[400]	Gaussian Naive Bayes
TEST 48	Image To Graph[200]	Gaussian Naive Bayes
TEST 49	Image To Graph[100]	Gaussian Naive Bayes
TEST 50	Image To Graph[50]	Gaussian Naive Bayes
TEST 51	Image To Graph[960]	Decision Tree
TEST 52	Image To Graph[400]	Decision Tree
TEST 53	Image To Graph[200]	Decision Tree
TEST 54	Image To Graph[100]	Decision Tree
TEST 55	Image To Graph[50]	Decision Tree
TEST 56	Image To Graph[960]	Random Forest
TEST 57	Image To Graph[400]	Random Forest
TEST 58	Image To Graph[200]	Random Forest
TEST 59	Image To Graph[100]	Random Forest
TEST 60	Image To Graph[50]	Random Forest
TEST 61	Daisy[400]	KNN
TEST 62	Daisy[200]	KNN
TEST 63	Daisy[100]	KNN
TEST 64	Daisy[50]	KNN
TEST 65	Daisy[400]	Gaussian Naive Bayes
TEST 66	Daisy[200]	Gaussian Naive Bayes
TEST 67	Daisy[100]	Gaussian Naive Bayes
TEST 68	Daisy[50]	Gaussian Naive Bayes
TEST 69	Daisy[400]	Decision Tree
TEST 70	Daisy[200]	Decision Tree
TEST 71	Daisy[100]	Decision Tree
TEST 72	Daisy[50]	Decision Tree
TEST 73	Daisy[400]	Random Forest
TEST 74	Daisy[200]	Random Forest
TEST 75	Daisy[100]	Random Forest
TEST 76	Daisy[50]	Random Forest
TEST 77	GIST[400] + PCA	KNN
TEST 78	DAISY[400] + PCA	KNN

After performing all of these tests, we are going to choose the best classification technique based on the relation between accuracy and time. This choice will be tested with the M0droid dataset to guarantee this correct behavior and we will also use this classifier with the packed applications to check if it is better to perform an unpacking process before or not. Besides, we have performed tests using two classifiers (GIST with daisy, Histogram with daisy) at the same time but we did not improve the accuracy and the time was high, so they are not considered and therefore, they are not represented in the following subsections.
Totally, we have performed 90 tests. We have repeated each test 5 times in order to guarantee a correct result by calculating the average of these 5 times.

6.2. Accuracy

6.2.1. Subtract Classification

After performing all the tests with the Malgenome dataset, the accuracy (%) achieved for the first approach was:

- TEST A: 80.75%
- TEST B: 73.78%
- TEST C: 27.91%

6.2.2. Extract Features and Classification

Regarding the Features Extraction+Classification approach, the accuracy (percentage 1) obtained is represented with the following table and it is shown graphically:

Image descriptors	KNN	Bayes	DecisionTree	RandomTree
GIST [960]	0.819131832797	0.659163987138	0.682475884244	0.739549839228
GIST [400]	0.800643086817	0.648713826367	0.688102893891	0.729099678457
GIST [200]	0.802250803859	0.590836012862	0.653536977492	0.735530546624
GIST [100]	0.759646302251	0.565112540193	0.643890675241	0.711414790997
GIST [50]	0.739549839228	0.466237942122	0.627813504823	0.699356913183
PCA-GIST[400]	0.818327974277			
HISTO[960]	0.738745980707	0.315112540193	0.627009646302	0.682475884244
HISTO[400]	0.759646302251	0.360128617363	0.628617363344	0.709807073955
HISTO[200]	0.771704180064	0.328778135048	0.629421221865	0.688102893891
HISTO[100]	0.741961414791	0.406752411576	0.59807073955	0.682475884244
HISTO[50]	0.66961414791	0.298231511254	0.610128617363	0.652733118971
imgToGr[960]	0.632636655949	0.632636655949	0.536173633441	0.630225080386
imgToGr[400]	0.647909967846	0.600482315113	0.547427652733	0.62459807074
imgToGr[200]	0.622186495177	0.545819935691	0.569935691318	0.631832797428
imgToGr[100]	0.603697749196	0.483118971061	0.553054662379	0.613344051447

imgToGr[50]	0.56270096463	0.426848874598	0.550643086817	0.608520900322
Daisy[400]	0.801446945338	0.595659163987	0.704180064309	0.764469453376
Daisy[200]	0.786977491961	0.601286173633	0.698553054662	0.748392282958
Daisy[100]	0.765273311897	0.475884244373	0.656752411576	0.724276527331
Daisy[50]	0.734726688103	0.467845659164	0.628617363344	0.720257234727
Daisy[400] + PCA	0.83038585209			

Table 12. Accuracy Features Extraction + Classification

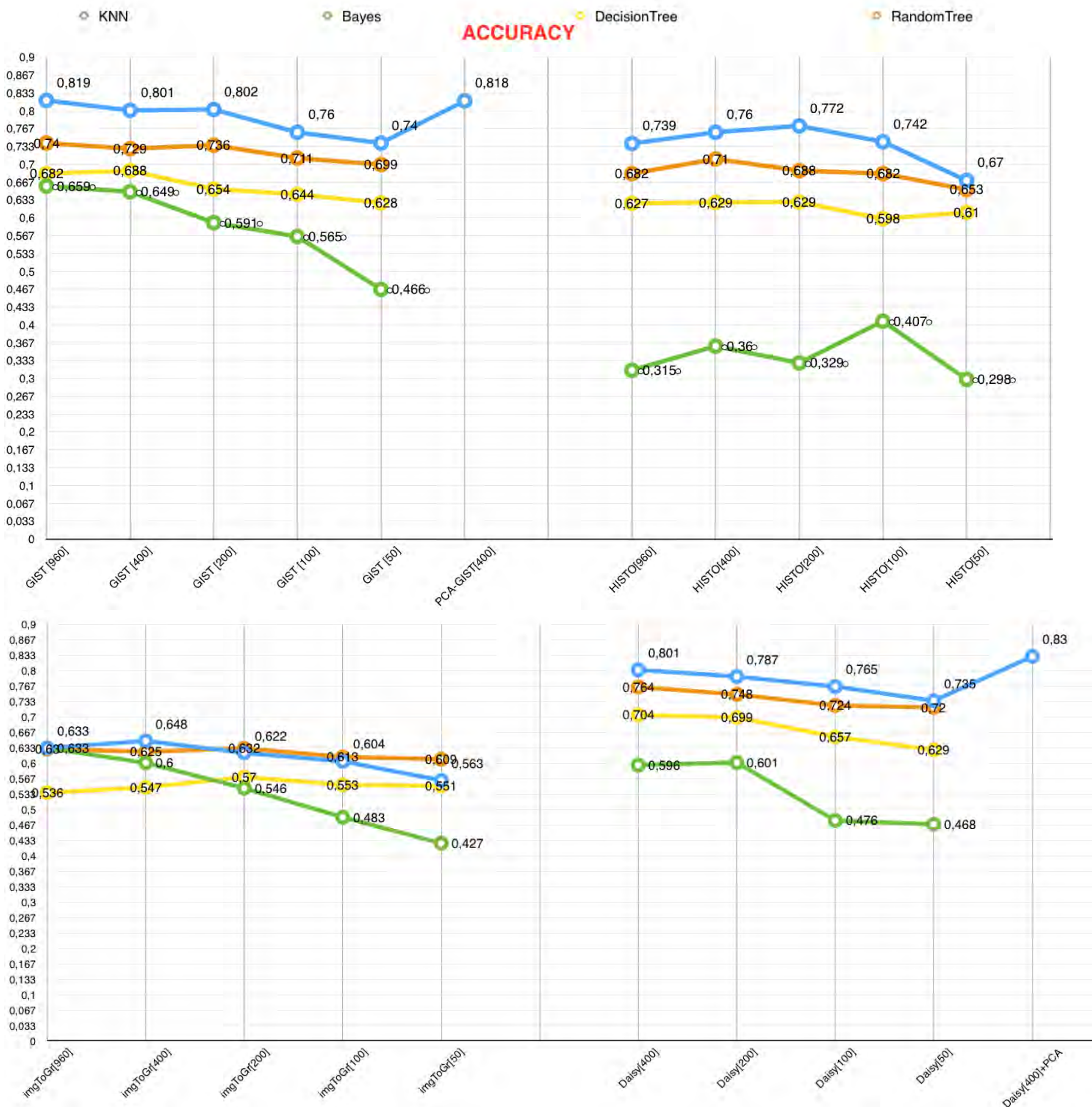


FIGURE 30. ACCURACY GRAPH FEATURES EXTRACTION + CLASSIFICATION

6.3. Time

6.3.1. Subtract Classification

After performing all the tests above, we have measured the total time for the first approach execution, getting the following results:

- TEST A: 151minutes with 58.434s
- TEST B: 232minutes with 2.659s
- TEST C: 26minutes with 56.107s

6.3.2. Extract Features and Classification

Regarding the Features Extraction+Classification approach, we have measured the time for features extraction, the training and testing time and the whole time needed for the program execution.

We have represented the results (in seconds) graphically for each classifier. The table with all of these measurements, can be found in the Annex III.

- KNN

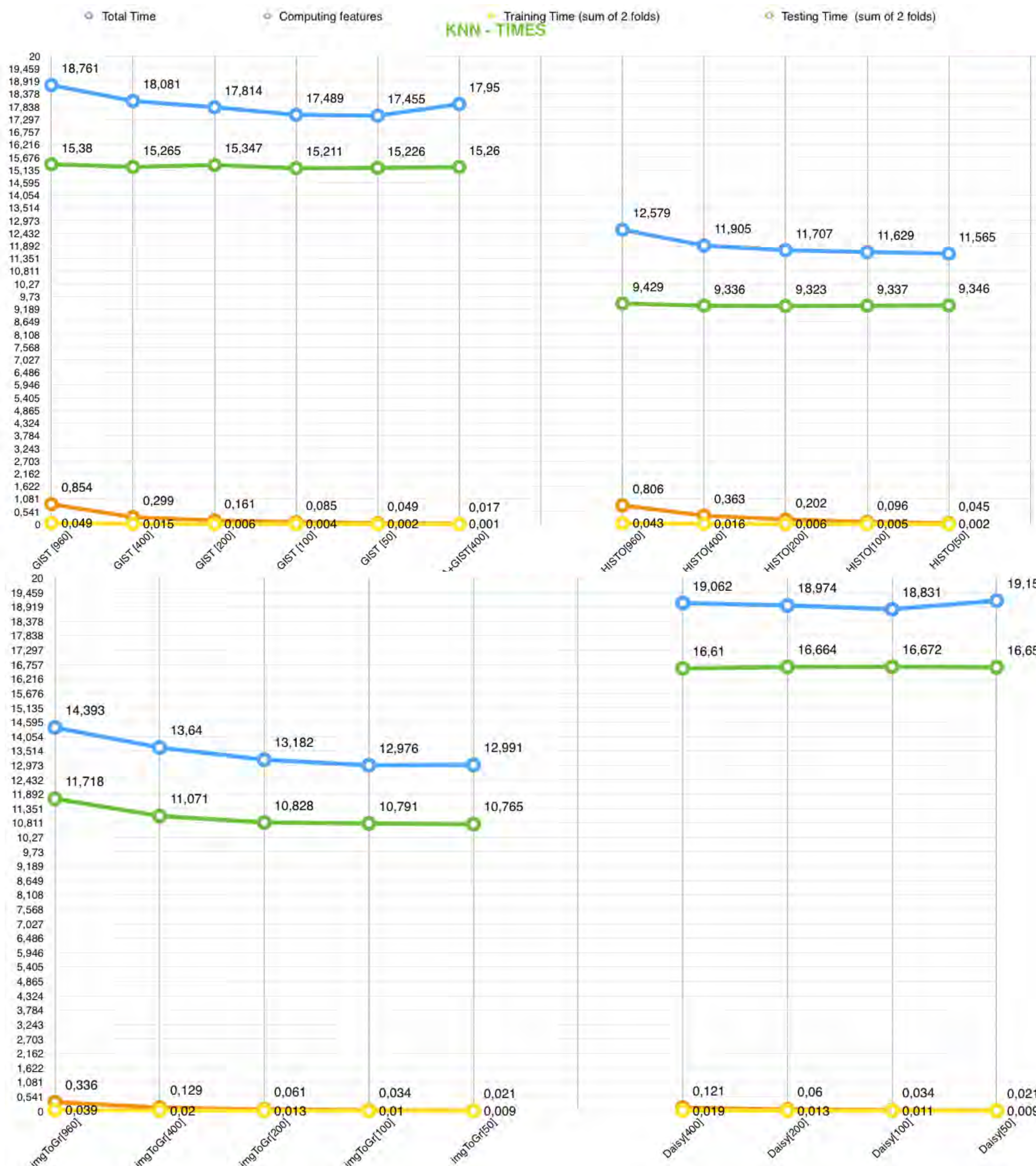


FIGURE 31. TIME GRAPH FEATURES EXTRACTION + KNN

- Gaussian Naive Bayes

FIGURE 32. TIME GRAPH FEATURES EXTRACTION + NAIVE BAYES

- Decision Tree

FIGURE 33. TIME GRAPH FEATURES EXTRACTION + DECISION TREE

- Random Forest

FIGURE 34. TIME GRAPH FEATURES EXTRACTION + RANDOM FOREST

Apart from these time measurements, the time used for the two-first phases, unpacking the application and converting classes.dex to an image, are: 1 minute with 1.137 seconds and 1 minute with 20.815 seconds, respectively.

6.4. Analysis of Results and Classifier Decision

6.4.1. Evaluation of Results

Considering the measurements obtained after performing the tests, we can deduce the conclusions explained above.

Regarding the accuracy:

- In the first classification strategy, the highest accuracy (80.75%) is reached using 80% of samples for training. This accuracy is reduced if we decrease the number of samples in the training set, with 60% the accuracy was 73.78% and with 1 sample for each family the accuracy was 27.91%, which does not fulfill the requirement *R03* so it must be discarded. We can observe that this strategy works well if the number of samples in the training set is high, this is due to the fact that if there are a lot of training samples there will be more comparisons between images so the computer will be able to differentiate better between families, it can extract a differential pattern with more detail.

- In the case of extracting image features and then using a machine learning classifier, the highest accuracy is reached with the KNN classifier, independently of the image descriptor used. The worst classifier is Naive Bayes, in most of the cases the accuracy is lower than 50%, it does not fulfill requirement *R03* so it is discarded. The rest of classifiers satisfy this requirement so they are considered. The best image descriptors are, in decreasing order, GIST, Daisy, Histogram and Image to Graph. However, if we apply PCA to Daisy with 400 values, we obtain the highest accuracy (83.04%) conversely to GIST (without PCA, 81.91%, and with PCA, 81.83%), meaning that the characteristics of GIST are sufficiently good to perform an accurate classification but they are similar so we cannot select the best ones. The reason obtained after this process is that the characteristics extracted with Daisy are so distinguishing that if we select the best ones with PCA and we apply KNN, the computer is able to divide the families and assign a class to an unknown sample based on the neighbors' features, which are analogous between them.

As a consequence of these two conclusions, we can establish that the most accurate strategy is extracting features with Daisy descriptor with 400 values retrieved, then applying PCA features selector to obtain the most differential characteristics and finally using KNN machine learning classifier to being able to distinguish between samples.

The confusion matrix obtained after applying the previous strategy (Daisy and KNN) is shown in *Figure 35*. In this figure we can see that most of the samples are well classified due to the diagonal of the matrix, the elements on the diagonal shows that the predicted values were correct (equal to the true label) and the elements outside this diagonal are the ones misclassified. Most of the elements stay on the diagonal. We can observe too that the values of the diagonal are really high, 1 or near 1 in lot of cases (100% success rate). The higher the diagonal values, the better, showing many correct guesses and the right behavior of the classification strategy.
If we compare *Figure 35* with *Figure 36*, which represents the worst classification (subtraction classification with 1 training sample) we can observe the differences and how the classification was done, showing that our measurements were right.

The rest of confusion matrices from all the tests performed can be found in Annex II.

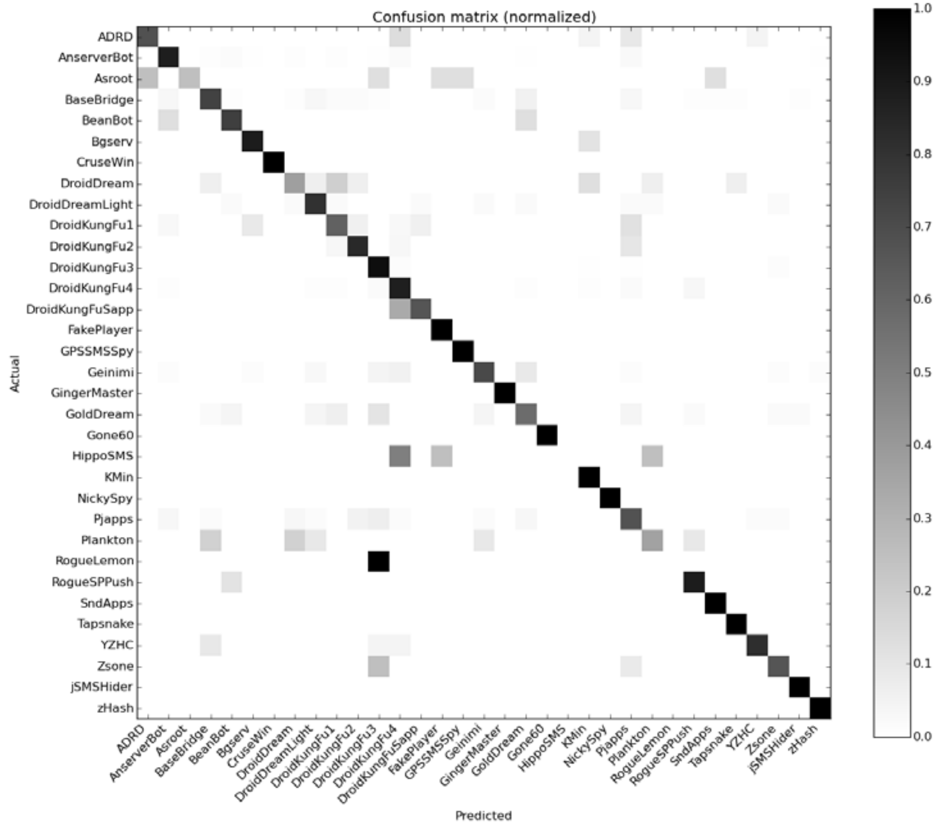


FIGURE 35. BEST CLASSIFICATION (DAISY + KNN) CONFUSION MATRIX

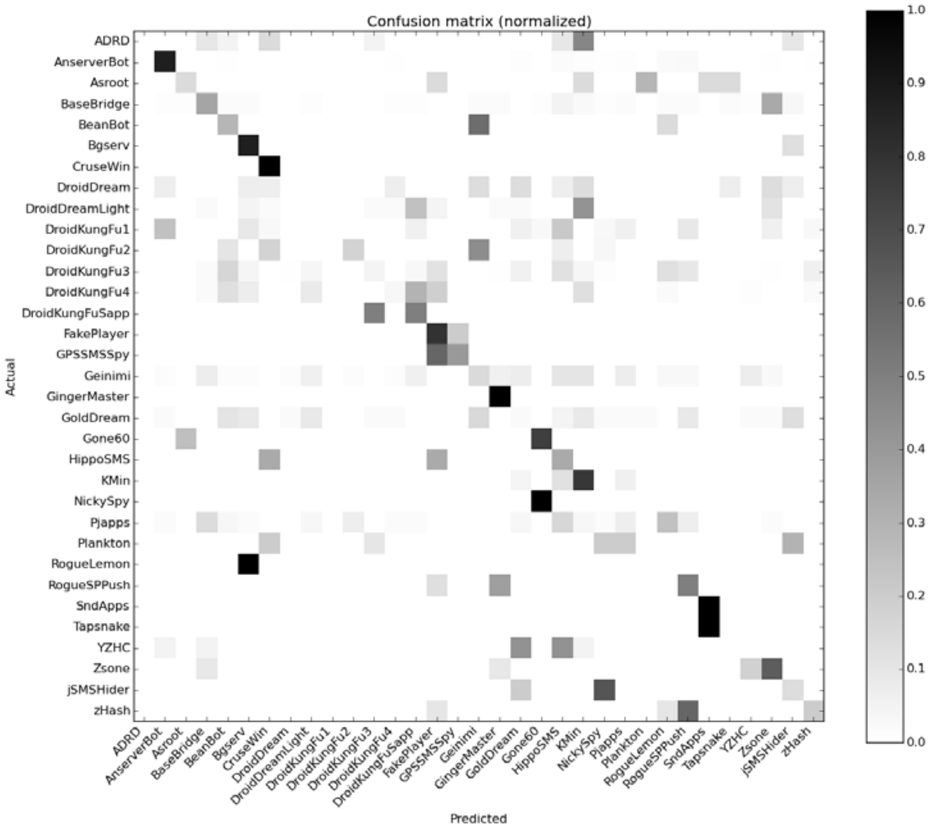


FIGURE 36. WORST CLASSIFICATION (SUBTRACT WITH 1) CONFUSION MATRIX

Regarding the time:

- The time used in the first strategy is really higher than the second strategy. To know if they fulfill the requirement *R02* we will compute the total time considering the time used for unpacking the app, converting classes.dex to an image and finally performing the classification. In the following table it is shown the time in hours for each classification technique but only with the strategies that fulfill the requirement related with the accuracy *(R03)*, commented previously (TEST C and most of tests with Naive Bayes Classifier are not considered).

	Time (hours) for 100,000 samples
TEST A	206.778
TEST B	314.054
TEST 1	3.588
TEST 2	3.573
TEST 3	3,.567
TEST 4	3.560
TEST 5	3.559
TEST 6	3.628
TEST 7	3.608
TEST 8	3.591
TEST 9	3.589
TEST 11	3.595
TEST 12	3.585
TEST 13	3.599
TEST 14	3.561
TEST 15	3.560
TEST 16	3.588
TEST 17	3.584
TEST 18	3.570
TEST 19	3.571
TEST 20	3.564
TEST 21	3.450
TEST 22	3.435
TEST 23	3.431

TEST 24	3.429
TEST 25	3.428
TEST 31	3.498
TEST 32	3.443
TEST 33	3.435
TEST 34	3.428
TEST 35	3.427
TEST 36	4.214
TEST 37	3.434
TEST 38	3.430
TEST 39	3.434
TEST 40	4.190
TEST 41	3.482
TEST 42	3.468
TEST 43	3.461
TEST 44	3.461
TEST 45	3.458
TEST 46	3.491
TEST 47	3.474
TEST 48	3.464
TEST 51	3.489
TEST 52	3.473
TEST 53	3.471
TEST 54	3.460
TEST 55	3.461
TEST 56	3.473
TEST 57	3.466
TEST 58	3.464
TEST 59	3.462
TEST 60	3.461
TEST 61	3.595
TEST 62	3.590

TEST 63	3.589
TEST 64	3.590
TEST 65	3.592
TEST 66	3.593
TEST 69	3.604
TEST 70	3.596
TEST 71	3.592
TEST 72	3.588
TEST 73	3.592
TEST 74	3.594
TEST 75	3.593
TEST 76	3.597
TEST 77	3.570
TEST 78	3.580

Table 13. Time for 100,000 samples (hours)

As we can see from *Table 13*, the tests performed with the first strategy consume a lot of time, they do not fulfill the requirement. That's why we predicted to do another strategy because the first tests performed showed that this were not the ideal behavior. All of the tests are related with the second approach and fulfill the requirement. As we can observe, most of the time is taken by the extracting features step, the classification process is really fast. The fastest descriptor is Histogram, followed by Image to Graph, Daisy and GIST. One remarkable detail of the classifiers is that, even though the classification is really fast, we can appreciate differences between them, for example, in the case of Decision Tree classifier, it has the fastest testing time but on the contrary, the training time is the highest. This is because it consumes lot of time building the tree (training time) but once it is built, the process of traversing the tree is really fast (testing time).

6.4.2. Final Decision

Considering the previous analysis, the first strategy does not fulfill the requirements so the best way for classifying malware converted to images is by extracting the characteristics of these images and then classifying them with a machine learning classifier.

The classification using subtraction of images technique has the disadvantage that it needs lots of samples to perform an accurate classification so it has to subtract the same images lots of times (for each unknown sample it subtract all the images of the training set with this sample) so it wastes so much time.

Regarding the second approach, it depends on the user needs to establish a final determination. If the user wants the fastest algorithm, we propose to use the Histogram with 50 values retrieved as an extracting features algorithm and the Decision Tree classifier; however the accuracy is only 61% so many samples will be misclassified.

If the user wants the most accurate algorithm, independently of the time, we propose to use Daisy descriptor with 400 values, PCA features selector and KNN classifier.

We have decided to select as the best classification technique the most accurate one (83.04%): Daisy with 400 values retrieved, PCA features selector and KNN classifier. There is a slightly difference in time between Daisy and GIST (0.01 hours difference) but we prefer to get the highest number of right guesses and invest a little bit more of time to achieve this. This technique is going to be used for classifying the samples of M0droid dataset in order to confirm this choice.

6.5. Final Results

6.5.1. Classification with M0droid Dataset

We have performed more tests with our final classifier. We have applied to M0droid dataset the process of unpacking an application, converting the *classes.dex* file to an image, extracting its characteristics with Daisy image descriptor with 400 values retrieved, selecting the best 25 values with PCA and classifying these images with KNN.

The time used for each process was the following:

- Unpacking the applications: 27.965 seconds
- Converting *classes.dex* to images: 1 minute with 1.027 seconds
- Daisy extraction features and KNN classification: 9.03 seconds

The accuracy obtained was 88%.

The obtained confusion matrix was the one above. We can see that our choice was right because it classifies well each family. It was able to distinguish between goodware and malware.

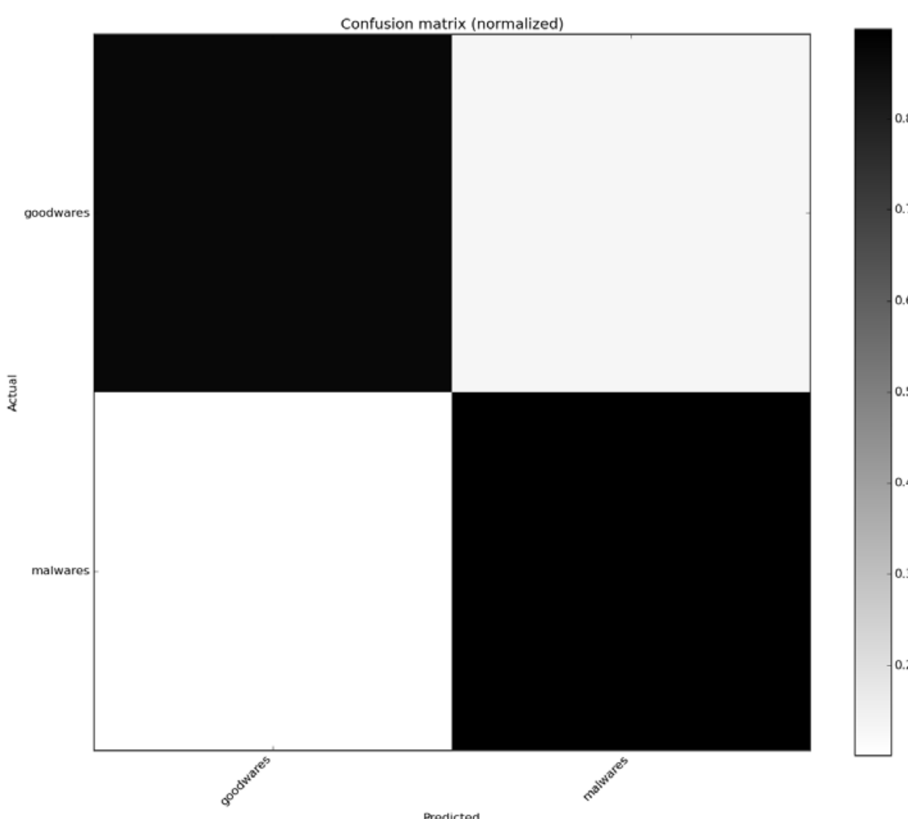


FIGURE 37. CONFUSION MATRIX M0DROID DATASET

6.5.2. Packed Applications Classification

Additionally, we have also used this dataset to perform a classification with packed applications. As we have mentioned during the whole project, our analysis was based on obtaining an accurate classification using images of malware. To obtain these images we have to unpack the application and get the *classes.dex* file. However, it consumes time so maybe its preferable to convert directly a packed application and do the classification.

The time spent for transforming directly a packed application to an image with the M0droid dataset was 1 minute with 41.317 seconds, which is more than extracting the files and then converting *classes.dex* (1 minute with 28.992 seconds). This is due to the size of an *.apk* file, as it contains more information, the time spent for converting it to an image is higher. Besides, the accuracy is reduced from 88% to 82.5% with a total classification time of 14.611 seconds, instead of 9.03 seconds achieved previously. The confusion matrix obtained with the packed applications is:

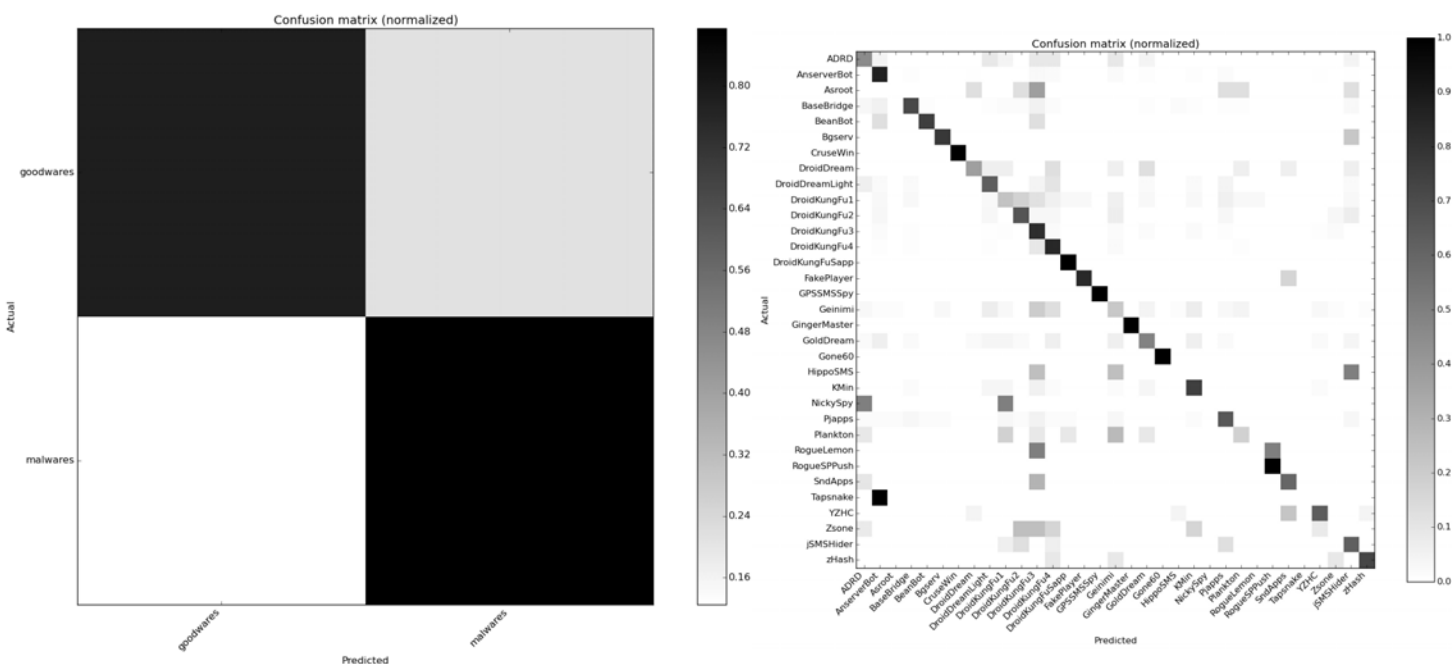


FIGURE 38. PACKED APPLICATIONS CONFUSION MATRICES

In *Figure 38*, we observe that some samples of goodware are classified as malware and vice versa, which is not desirable. There is a considerable number of samples classified out of the diagonal. The same thing happens with the Malgenome dataset, the accuracy is decreased and the time is increased as well (69.29% of accuracy, 4 minutes with 17.534 for transforming to images and 39.79 seconds for the classification).

We can conclude that it is preferable to unpack the application than using the APK file directly. However, if the malware is encrypted, with this approach we can solve this problem.

7. Project Design and Budget

7.1. Gantt Chart

In this section we present the following Gantt Chart diagram which contains the different project tasks, interconnections and dependencies.

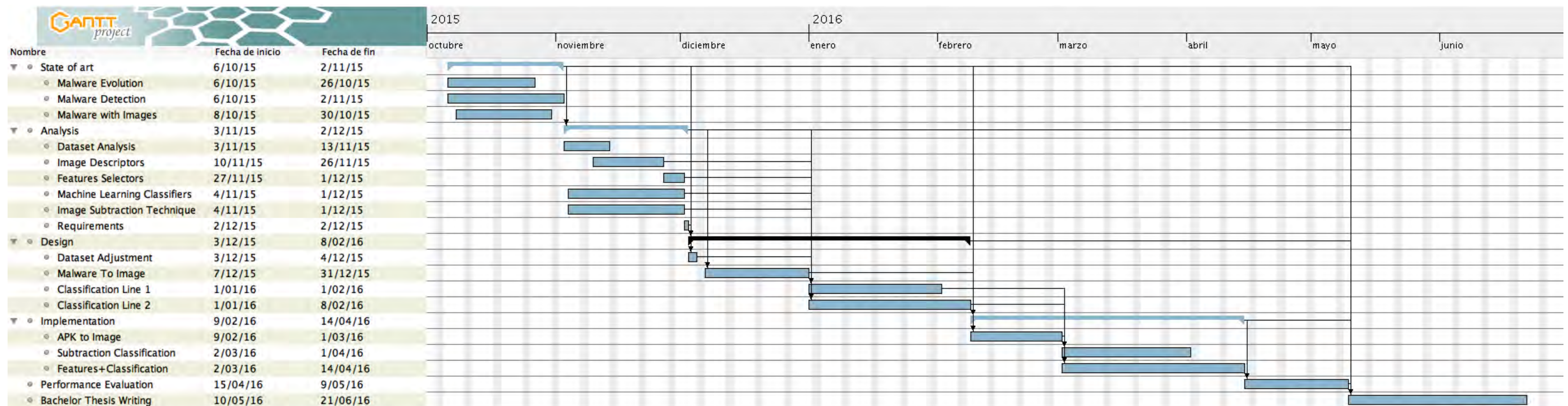


FIGURE 39. GANTT CHART

7.2. Estimated Costs

This section covers all the costs and needs that are contemplated during the project development. They are calculated taking into account the whole period of the project.

7.2.1. Hardware Equipment

Only the hardware elements directly related with the project consecution are included in the following table. The expenses are calculated considering 21% of the price of the item.

Item	Price (€)	Acquisition Date	Total Cost (with expenses) (€)
iMac (21.5 inch, mid 2011)	513.50	June-2011	650.00
SanDisk USB 3.0 32GB	7.347	September-2015	9.30
		TOTAL	659.30

Table 14. Budget - Hardware Equipment

7.2.2. Software Licenses

The software components used in this project are the following ones. The expenses are calculated considering 21% of the price of the item.

Item	Price (€)	Acquisition Date	Total Cost (with expenses) (€)
Pages for Mac	15.78	September - 2015	19.99
VMWare Fusion 8	67.73	October-2015	81.95
Python	0.00	October-2015	0.00
Python External Libraries	0.00	November-2015	0.00
Ubuntu OS 15.10	0.00	October-2015	0.00
Malgenome Dataset	0.00	November-2015	0.00
M0droid Dataset	0.00	November-2015	0.00
		TOTAL	101.94

Table 15. Budget - Software Licenses

7.2.3. Human Resources

Salaries are detailed according to each role needed for this thesis. There are three roles: project manager, analyst and programmer. The former is performed by the thesis tutor, Pedro Peris-López and the later (analyst and programmer) are performed by the thesis author, Raquel Tabuyo Benito.

Project Phase	Role	Salary (€/hour)	Hours/day	Total Days	Total Cost (€)
Investigation (State of art)	Analyst	25	5	20	2,500
Analysis	Analyst	25	5	22	2,750
Design	Analyst	25	5	48	6,000
Implementation	Programmer	20	5	48	4,800
Performance Evaluation	Programmer	20	5	17	1,700
Thesis Writing	Analyst	25	5	31	3,875
Project Management	Project Manager	35	2	90	6,300
				TOTAL	27,925

Table 16. Budget - Human Resources

7.2.4. Direct Costs

The cost of all the elements of the project is shown in the following table:

Description	Total Cost
Hardware Equipment	659.3
Software Components	101.94
Human Resources	27,925
TOTAL	28,686.24

Table 17. Budget - Direct Costs

7.2.5. Indirect Costs

The indirect costs derived from this project are calculated considering 2% margin of the direct costs:

Direct Costs	Indirect Margin	Total Cost
28,686.24	2 %	573.73

Table 18. Budget - Indirect Costs

7.2.6. Benefits

As this thesis is an experimental project with educational purposes, it is a nonprofit project. Therefore, the total benefit is 0.00€.

7.2.7. Risks

We are considering a risk margin of 15% of the direct costs:

Direct Costs	Indirect Margin	Total Cost
28,686.24	15 %	4,302.94

Table 19. Budget - Risks

7.2.8. Grand Total

The total cost of this project is calculated in the following table:

Description	Total Cost
Direct costs	28,686.24
Indirect costs	573.73
Benefits	0.00
Risks	4,302.94
TOTAL	33,562.91

Table 20. Budget - Grand Total

8. Conclusions and Future Work

We have proposed two ways of Android malware classification based on converting an application to a PNG image and performing the classification.

The first approach was an innovative strategy based on subtracting images from the testing set with images of the training set, however it does not fulfill the requirements of time because this strategy requires lot of samples in the training set to perform an accurate classification. So at the end, we have to do the same process (subtracting images) for each image of the testing set, producing a wasting of time.

As the first approach requires a lot of time, we decided to develop another strategy based on using an image descriptor and a machine learning classifier. With this technique we have fulfilled the requirements. The limit was 50% of accuracy and 10 hours for 100,000 samples (converting to images and classifying) and we have achieved an accuracy more than 80% in less than 4 hours. Specifically, the two datasets used are classified with an accuracy of 83.04% in 18.396 seconds (Malgenome dataset) and with an accuracy of 88% in 9.03 seconds (M0droid dataset). The classification was made using Daisy image descriptor with 400 values retrieved, the selection of the best 25 values with PCA features selector and finally, the classification with KNN machine learning classifier. To this process we have to add the time of unpacking the application and the conversion to image process (1 minute with 1.137 seconds and 1 minute with 20.815 seconds, respectively, for Malgenome; 27.965 seconds and 1 minute with 1.027 seconds for M0droid). The accuracy for packed applications drops and the time is increased, however if the malware is encrypted and it is not possible to unpack the application, we can analyze the packed application with this process.

As a conclusion we can deduce that due to the classification relies on the information obtained from the textures of the PNG images, it is not required to execute the code nor any disassembly process, making this detection strategy 40 times faster than the traditional ones because we only need 400 values instead of the 65,000 elements required in distribution based analysis. Besides, this detection technique avoids obfuscation and malware encryption.

However, the main drawback for this strategy is that if the attacker knows that it is used as a detection technique, he can add not relevant data (zero-values) or relocate sections of the Android file in order to generate a different image in this process.

The future lines can be a software development project that uses this idea as the basis or the improvement of the first approach in order to obtain a classifier that does not need a huge number of samples for the training set to reduce the classification time.

9. References

[1] Suarez-Tangil, Guillermo, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. "Evolution, Detection and Analysis of Malware for Smart Devices." *IEEE Communications Surveys & Tutorials IEEE Commun.* Surv. Tutorials 16, no. 2 (2014): 961-87. doi:10.1109/surv. 2013.101613.00077.

[2] Cocotas, Alex. "Smartphone Sales Will Reach Nearly 1.6 Billion Units By 2016." Business Insider. 2012. Accessed October 6, 2015. http://www.businessinsider.com/smartphone-sales-will-reach-almost-16-billion-units-by-2016-2012-2

[3] "Android Just Achieved Something It Will Take Apple Years to Do." Accessed October 6, 2015. http://uk.businessinsider.com/android-1-billion-shipments-2014-strategy-analytics-2015-2

[4] "Symantec - 2016 Internet Security Report." Symantec - 2016 Internet Security Report. Accessed October 7, 2015. https://resource.elq.symantec.com/istr-vol21-en

[5] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[6] C. Fleizach, M. Liljenstam, P. Johansson, G. Voelker, and A. Mehes, "Can you infect me now?: malware propagation in mobile phone networks," in *Proc. 2007 ACM workshop on Recurring malcode*. ACM, 2007, pp. 61–68.

[7] R. Verdult and F. Kooman, "Practical attacks on nfc enabled cell phones," in *3rd Int. Workshop on Near Field Commun. (NFC)*, February 2011, pp. 77–82.

[8] Samsung. Accessed October 10, 2016. http://www.samsung.com/es/tv/

[9] C. Xiang, F. Binxing, Y. Lihua, L. Xiaoyi, and Z. Tianning, "Andbot: towards advanced mobile botnets," in Proc. 4th USENIX conf. on Large-scale exploits and emergent threats, ser. LEET'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–11.

[10] Yajin Zhou and Xuxian Jiang, "An Analysis of the AnserverBot Trojan". September 25, 2011. Accessed February 3, 2016. https://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf

[11] Kramer, S, "Rage against the cage," 2010.

[12] Masaki Suenaga, "Android.Opfake In-Depth". 2012. Accessed October 13, 2015. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/android_opfake_in_depth.pdf

[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. 33rd IEEE Symp. Security and Privacy (Oakland 2012)*, May 2012.

[14] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter application communication in android," in *Proc. 9th int. conf. on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[15] Grayson Milbourne and Armando Orozco. "An In-depth Look at the Evolution of Android Malware". August 2012. Accessed October 26, 2015. http://www.brightcloud.com/pdf/Android-Malware-Exposed.pdf

[16] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *Information Security*, ser. Lecture Notes in Computer Science, M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, Eds. Springer Berlin / Heidelberg, 2011, vol. 6531, pp. 346–360.

[17] Felt, Adrienne Porter and Chin, Erika and Hanna, Steve and Song, Dawn and Wagner, David, “Android permissions demystified,” in *Proc. 18th ACM conf. on Computer and commun. security*. ACM, 2011, pp. 627–638.

[18] A. P. Felt, K. Greenwood, and D. Wagner, “The effectiveness of application permissions,” in *Proc. 2nd USENIX conf. on Web application development*, ser. WebApps’11. USENIX Association, 2011, pp. 7–7.

[19] E. Chin, A. Felt, K. Greenwood, and D. Wagner, “Analyzing inter application communication in android,” in *Proc. 9th int. conf. on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[20] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva, “Poster: Role based access control for android (rbaca),” Boston University, MA USA, Tech. Rep., 2012.

[21] X. Ni, Z. Yang, X. Bai, A. C. Champion, and D. Xuan, “Diffuser: Differentiated user access control on smartphones,” in *IEEE 6th Int. Conf. Mobile Adhoc and Sensor Systems, 2009. MASS’09.*. IEEE, 2009, pp. 1012–1017.

[22] G. Russello, M. Conti, B. Crispo, and E. Fernandes, “Moses: supporting operation modes on smartphones,” in *Proc. 17th ACM symp. on Access Control Models and Technologies*, ser. SACMAT ’12. New York, NY, USA: ACM, 2012, pp. 3–12.

[23] C. Mulliner, G. Vigna, D. Dagon, and W. Lee, “Using labeling to prevent crossservice attacks against smart phones,” in *Detection of Intrusions and Malware and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, R. Bschkes and P. Laskov, Eds. Springer Berlin Heidelberg, 2006, vol. 4064, pp. 91–108.

[24] N. Husted, H. Sa¨ıdi, and A. Gehani, “Smartphone security limitations: conflicting traditions,” in *Proc. 2011 Workshop on Governance of Technology, Information, and Policies*, ser. GTIP ’11. New York, NY, USA: ACM, 2011, pp. 5–12.

[25] M. Conti, V. Nguyen, and B. Crispo, “Crepe: Context-related policy enforcement for android,” *Information Security*, pp. 331–345, 2011.

[26] M. Knappmeyer, S. L. Kiani, E. S. Reetz, N. Baker, and R. Tonjes, “Survey of context provisioning middleware,” *IEEE Commun. Surveys & Tutorials*, vol. 15, no. 3, pp. 1492–1519, 2013.

[27] A.-D. Schmidt, “Detection of smartphone malware,” Ph.D. dissertation, Universitats-bibliothek, 2011.

[28] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “ “andromaly”: a behavioral malware detection framework for android devices,” *J. of Intelligent Information Systems*, vol. 38, pp. 161–190, 2012.

[29] F. Shahzad, M. Akbar, S. Khan, and M. Farooq, “Tstructdroid: Real time malware detection using in-execution dynamic analysis of kernel process control blocks on android,” National University of Computer & Emerging Sciences, Islamabad, Pakistan, Tech. Rep., 2013.

[30] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in Proc. 6th int. conf. on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security, ser. MMMACNS'12. Springer-Verlag, 2012, pp. 240–253.

[31] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior based malware detection system for android," in *1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[32] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. StypRekowsky, "Appguard —-real–time policy enforcement for third party applications," Universitats- und Landesbibliothek, Postfach 151141, 66041 Saarbracken, Tech. Rep., 2012. Accessed October 21, 2015. http://scidok.sulb.uni-saarland.de/volltexte/2012/4902

[33] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, "Secloud: A cloud-based comprehensive and lightweight security solution for smartphones," *Computers & Security*, 2013.

[34] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proc. 26th Annu. Computer Security Applications Conf.*, 2010, pp. 347–356.

[35] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003.

[36] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *5th Int. Conf. on Malicious and Unwanted Software (MALWARE 2010)*. IEEE, 2010, pp. 55–62.

[37] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proc. 21st USENIX conf. on Security symp.*. USENIX Association, 2012, pp. 29–29.

[38] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proc. 3rd ACM conference on Data and application security and privacy*. ACM, 2013, pp. 209– 220.

[39] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proc. 2nd ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93– 104.

[40] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proc. 10th int. conf. on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[41] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. B. Alis, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, 2013, in Press.

[42] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. of CCS 2010*, A. Keromytis and V. Shmatikov, Eds. ACM Press, Oct. 2010, pp. 559–72.

[43] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: statically vetting android apps for component hijacking vulnerabilities,” in *Proc. 2012 ACM conf. on Computer and communications security*. ACM, 2012, pp. 229–240.

[44] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in *Proc. 19th Annu. Symp. on Network and Distributed System Security*, 2012.

[45] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K. Saul and Geoffrey M. Voelker, “eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones”. Accessed February 19, 2016. https://cseweb.ucsd.edu/~voelker/pubs/edoctor-nsdi13.pdf

[47] S. Rosen, Z. Qian, and Z. M. Mao, “Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users,” in *Proc. 3rd ACM conference on Data and application security and privacy*. ACM, 2013, pp. 221–232.

[48] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *Proc. 9th USENIX conf. on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–6.

[49] L. Nataraj, S. Karthikeyan, G. Jacob, B. S. Manjunath, "Malware Images: Visualization and Automatic Classification." Malware Images. Accessed October 25, 2015. http://dl.acm.org/citation.cfm?id=2016908

[50] Douze, Matthijs, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. "Evaluation of GIST Descriptors for Web-scale Image Search." *Proceeding of the ACM International Conference on Image and Video Retrieval - CIVR '09*, 2009. Accessed October 24, 2015. doi:10.1145/1646396.1646421.

[51] "Android.Adrd." Symantec. Accessed November 3, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-021514-4954-99

[52] Yajin Zhou and Xuxian Jiang, “An Analysis of the AnserverBot Trojan”. September 25, 2011. Accessed November 3, 2015. https://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf

[53] Zhou, Yajin, and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution." *2012 IEEE Symposium on Security and Privacy,* 2012. doi:10.1109/sp.2012.16.

[54] "Android.Basebridge." Symantec. Accessed November 3, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99

[55] Xuxian Jiang, ”BeanBot." NC State University. Accessed February 29, 2016. https://www.csc.ncsu.edu/faculty/jiang/BeanBot/

[56] "Android.Bgserv." Symantec. Accessed November 3, 2015, 2016. https://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99

[57] "SMS Spying Android Trojan Triggered by Keywords - Help Net Security." Help Net Security. 2011. Accessed November 3, 2015. https://www.helpnetsecurity.com/2011/08/08/sms-spying-android-trojan-triggered-by-keywords/

[58] "Android.Crusewind." Symantec. Accessed November 3, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-070301-5702-99

[59] "Android.Dogowar." Symantec. Accessed November 4, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-081510-4323-99

[60] Xuxian Jiang, “Security Alert: New Android Malware -- *DroidCoupon* -- Found in Alternative Android Markets” NC State University. Accessed November 4, 2015 https://www.csc.ncsu.edu/faculty/jiang/DroidCoupon/

[61] Xuxian Jiang, “Security Alert: New Root-Capable *DroidDeluxe* Malware Found in Alternative Android Markets” NC State University. Accessed November 4, 2015. https://www.csc.ncsu.edu/faculty/jiang/DroidDeluxe/

[62] "Technical Analysis | Lookout Blog." Lookout Blog RSS. Accessed November 4, 2015. https://blog.lookout.com/droiddream/

[63] Dunham, Ken, Shane Hartman, Jose Andre. Morales, Manu Quintans, and Tim Strazzere. *Android Malware and Analysis*.

[64] Xuxian Jiang, “Security Alert: New Sophisticated Android Malware *DroidKungFu* Found in Alternative Chinese App Markets” NC State University. Accessed November 4, 2015.. https://www.csc.ncsu.edu/faculty/jiang/DroidKungFu/

[65] Xuxian Jiang, “Security Alert: New *DroidKungFu* Variant -- AGAIN! -- Found in Alternative Android Markets” NC State University. Accessed November 4, 2015..https://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/

[66] Zhou, Yajin, and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution." 2012 *IEEE Symposium on Security and Privacy*, 2012. doi:10.1109/sp.2012.16.

[67] David Korczynski, “ClusTheDroid: Clustering Android Malware”. 4 March 2015. Information Security Group Royal Holloway University of London. Accessed November 5, 2015.. https://www.ma.rhul.ac.uk/static/techrep/2015/RHUL-MA-2015-1.pdf

[68] "Android Threat Set to Trigger On the End of Days, or the Day's End." Symantec Security Response. Accessed November 5, 2015. http://www.symantec.com/connect/blogs/android-threat-set-trigger-end-days-or-day-s-end

[69] "Security Alert: Fake Netflix App Aids Phishing | Lookout Blog." Lookout Blog RSS. Accessed November 5, 2015. https://blog.lookout.com/blog/2011/10/13/security-alert-fake-netflix-app-aids-phishing/

[70] Xuxian Jiang, “Security Alert: New Android Malware -- *GoldDream* -- Found in Alternative App Markets” NC State University. Accessed November 5, 2015. https://www.csc.ncsu.edu/faculty/jiang/GoldDream/

[71] "Android.Golddream." Symantec. Accessed November 5, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-070608-4139-99

[72] "Android.Gonesixty." Symantec. Accessed November 6, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2011-093001-2649-99

[73] Jiang Xuxian, and Yajin Zhou. Android Malware. Dordrecht: Springer, 2013.

[74] "Beta Version of Spytool App for Android Steals SMS Messages - TrendLabs Security Intelligence Blog." TrendLabs Security Intelligence Blog. 2012. Accessed November 6, 2015. http://blog.trendmicro.com/trendlabs-security-intelligence/beta-version-of-spytool-app-for-android-steals-sms-messages/

[75] "Security Alert: Malware Found Targeting Custom ROMs (jSMSHider) | Lookout Blog." Lookout Blog RSS. Accessed November 6, 2015. https://blog.lookout.com/blog/2011/06/15/security-alert-malware-found-targeting-custom-roms-jsmshider/

[76] "The Growing Threat of Mobile Malware: Top Android Malware Families of 2012 - Quick Heal Technologies Security Blog | Latest Computer Security News, Tips, and Advice." Quick Heal Technologies Security Blog Latest Computer Security News Tips and Advice. 2013. Accessed November 6, 2015. http://blogs.quickheal.com/the-growing-threat-of-mobile-malware-top-android-malware-families-of-2012/

[77] Xuxian Jiang, "Security Alert: New *NickiBot* Spyware Found in Alternative Android Markets" NC State University. Accessed November 7, 2015.https://www.csc.ncsu.edu/faculty/jiang/NickiBot/

[78] "Android.Nickispy." Symantec. Accessed March 06, 2016. https://www.symantec.com/security_response/writeup.jsp?docid=2011-072714-3613-99

[79] Xuxian Jiang, "Security Alert: New Rogue App *RogueLemon* Found in Alternative Chinese Android Markets" NC State University. Symantec. Accessed November 7, 2015. https://www.csc.ncsu.edu/faculty/jiang/RogueLemon/

[80] Xuxian Jiang, "New Rogue Android App -- *RogueSPPush* -- Found in Alternative Android Markets" NC State University. Accessed November 7, 2015. https://www.csc.ncsu.edu/faculty/jiang/RogueSPPush/

[81] Xuxian Jiang, "Questionable Android Apps -- *SndApps* -- Found and Removed from Official Android Market" NC State University. Accessed November 7, 2015. https://www.csc.ncsu.edu/faculty/jiang/SndApps/

[82] "News from the Lab Archive : January 2004 to September 2015." News from the Lab Archive : January 2004 to September 2015. Accessed November 8, 2015. https://www.f-secure.com/weblog/archives/00002236.html

[83] "Android.Tapsnake." Symantec. Accessed November 8, 2015. https://www.symantec.com/security_response/writeup.jsp?docid=2010-081214-2657-99

[84] "AndroidOS.Tapsnake: Watching Your Every Move." Symantec Security Response. Accessed November 8, 2015. http://www.symantec.com/connect/blogs/androidostapsnake-watching-your-every-move

[85] Xuxian Jiang, "Security Alert: New Android SMS Trojan -- YZHCSMS -- Found in Official Android Market and Alternative Markets" NC State University. Accessed November 8, 2015. https://www.csc.ncsu.edu/faculty/jiang/YZHCSMS/

[86] "Security Alert: Zsone Trojan Found in Android Market | Lookout Blog." Lookout Blog RSS. Accessed November 8, 2015. https://blog.lookout.com/blog/2011/05/11/security-alert-zsone-trojan-found-in-android-market/

[87] "Security News." Security News. Accessed November 8, 2015. http://www.pctools.com/security-news/what-is-malware/

[88] E. Chien, “Motivations of Recent Android Malware”. 2011. Accessed November 9, 2015. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/motivations_of_recent_android_malware.pdf

[89] “Mobile malware: A network view” Black Hat Mobile Security Summit – London 2015. Accessed November 9, 2015. https://www.blackhat.com/docs/ldn-15/materials/london-15-McNamee-Mobile-Malware-A-Network-View-wp.pdf

[90] "The Rise of Android Ransomware - We Live Security." Accessed November 9, 2015. http://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf

[91] "Ransomware." - Definition. Accessed November 9, 2015. http://www.trendmicro.com/vinfo/us/security/definition/ransomware

[92] Damshenas, Mohsen, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Ramlan Mahmud. M0Droid: An Android Behavioral-Based Malware Detection Model. *Journal of Information Privacy and Security* 11, no. 3 (2015): 141-57. doi:10.1080/15536548.2015.1073510.

[93] Zhou, Yajin, and Xuxian Jiang. “Dissecting Android Malware: Characterization and Evolution”. *2012 IEEE Symposium on Security and Privacy*, 2012. doi:10.1109/sp.2012.16.

[94] A. Oliva and A. Torralba. Modeling the shape of the scene: a holistic representation of the spatial envelope. IJCV, 42(3):145–175, 2001.

[95] Douze, Matthijs, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg, and Cordelia Schmid. "Evaluation of GIST Descriptors for Web-scale Image Search." *Proceeding of the ACM International Conference on Image and Video Retrieval - CIVR '09,* 2009. doi:10.1145/1646396.1646421.

[96] A. Martínez Retenaga, “Android Malware situation”. Spanish National Cybersecurity Institute. February 2015. Accessed November 13, 2015. https://www.incibe.es/extfrontinteco/img/File/intecocert/EstudiosInformes/android_malware_situation.pdf

[97] Somarriba, Oscar, Urko Zurutuza, Roberto Uribeetxeberria, Laurent Delosières, and Simin Nadjm-Tehrani. "Detection and Visualization of Android Malware Behavior." *Journal of Electrical and Computer Engineering 2016* (2016): 1-17. doi:10.1155/2016/8034967.

[98] "Histograms Introduction." www.tutorialspoint.com. Accessed November 10, 2015. http://www.tutorialspoint.com/dip/histograms_introduction.htm

[99] "User Guide." User Guide: Contents — Scikit-learn 0.12 Documentation. Accessed November 13, 2015. http://www.math.unipd.it/~aiolli/corsi/1213/aa/user_guide-0.12-git.pdf

[100] "Dense DAISY Feature Description." Dense DAISY Feature Description — Skimage V0.12dev Docs. Accessed November 15, 2015. http://scikit-image.org/docs/dev/auto_examples/plot_daisy.html

[101] Lowe, David G. "Distinctive Image Features from Scale-Invariant Keypoints." International Journal of Computer Vision 60, no. 2 (2004): 91-110. doi:10.1023/b:visi.0000029664.99615.94.

[102] Hu, Yu-Chen, “International Journal of Image Processing (IJIP)” Book: 2009 Volume 3, Issue 4. Publishing Date: 31-08-2009. ISSN (Online): 1985 - 2304. Accessed March 15, 2016 http://www.cscjournals.org/download/issuearchive/IJIP/Volume3/IJIP_V3_I4.pdf#page=16

[103] Wu, Jian, Zhiming Cui, Victor S. Sheng, Pengpeng Zhao, Dongliang Su, and Shengrong Gong. "A Comparative Study of SIFT and Its Variants." *Measurement Science Review* 13, no. 3 (2013). doi:10.2478/msr-2013-0021.

[104] S. O'Hara and B. A. Draper, "Introduction to the Bag Of Features Paradigm For Image Classification and Retrieval". 17 January 2011. Accessed November 20, 2015. https://arxiv.org/pdf/1101.3354.pdf

[105] D. Aldavert, A. Ramisa, R. Lopez de Mantaras and R. Toledo. "Real-Time Object Segmentation Using a Bag of Features Approach" Accessed November 21, 2015. http://www.iiia.csic.es/~mantaras/CCIA2010.pdf

[106] Kobayashi, Takumi. "BFO Meets HOG: Feature Extraction Based on Histograms of Oriented P.d.f. Gradients for Image Classification." *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013. doi:10.1109/cvpr.2013.102.

[107] Yang, Hongsheng, Wen-Yan Lin, and Jiangbo Lu. "DAISY Filter Flow: A Generalized Discrete Approach to Dense Correspondences." *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014. doi:10.1109/cvpr.2014.435.

[108] Tola, E., V. Lepetit, and P. Fua. "DAISY: An Efficient Dense Descriptor Applied to Wide-Baseline Stereo." *IEEE Transactions on Pattern Analysis and Machine Intelligence IEEE Trans. Pattern Anal. Mach. Intell.* 32, no. 5 (2010): 815-30. doi:10.1109/tpami.2009.77.

[109] B. A. Draper , K. Baek , M. Stewart Bartlett , J. Ross Beveridge, "Recognizing Faces with PCA and ICA" Accessed November 28, 2015. http://www.face-rec.org/algorithms/comparisons/draper_cviu.pdf

[110] Boiman, Oren, Eli Shechtman, and Michal Irani. "In Defense of Nearest-Neighbor Based Image Classification." *2008 IEEE Conference on Computer Vision and Pattern Recognition*, 2008. doi:10.1109/cvpr.2008.4587598.

[111] "Face Recognition." Final Project Writeup. Accessed March 16, 2016. http://cs.brown.edu/courses/csci1290/2011/results/final/amf1/

[112] "Facial age estimation". PCR. Accessed March 16, 2016. http://www.dia.fi.upm.es/~pcr/attributes.html

[113] J. Kim, Byung-Soo Kim, S. Savarese, "Comparing Image Classification Methods: K-Nearest-Neighbor and Support-Vector-Machine". Accessed November 5, 2015. http://www.wseas.us/e-library/conferences/2012/CambridgeUSA/MATHCC/MATHCC-18.pdf

[114] Lindsay I Smith, "A tutorial on Principal Components Analysis". February 26, 2002. Accessed November 7, 2015. http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf

[115] T.M. Mitchell, "Chapter 3: Generative and Discriminate Classifiers: Naive Bayes and Logistic Regression"Machine Learning, McGraw Hill.

[116] Sanches, João Miguel., Luisa Micó, and Jaime S. Cardoso. *Pattern Recognition and Image Analysis: 6th Iberian Conference, IbPRIA 2013, Funchal, Madeira, Portugal, June 5-7, 2013: Proceedings*.

[117] "1.10. Decision Trees." 1.10. Decision Trees — Scikit-learn 0.17.1 Documentation. Accessed November 15, 2015. http://scikit-learn.org/stable/modules/tree.html#tree

[118] A. Mukhtarov, S. Porshnev, V. Zuzin, A. Bobkova and V. Bobkov, “The Study of Applicability of the Decision Tree Method for Contouring of the Left Ventricle Area in Echographic Video Data”.

[119] Tom M. Mitchell, “Gaussian Naïve Bayes, and Logistic Regression” Carnegie Mellon University. January 25, 2010. Accessed November 23, 2015. http://www.cs.cmu.edu/~epxing/Class/10701-10s/Lecture/lecture5.pdf

[120] “Chapter 11: Image Processing - Classification”. Accessed November 28, 2015. http://www.jars1974.net/pdf/12_Chapter11.pdf

[121] Kun-Che Lu and Don-Lin Yang, “Image Processing and Image Mining using Decision Trees” Feng Chia University. July 17, 2008. Accessed November 29, 2015. http://www.iis.sinica.edu.tw/page/jise/2009/200907_02.pdf

[122] A. Liaw and M. Wiener, “Classification and Regression by randomForest” Vol. 2/3, December 2002.

[123] "Random Forests Algorithm." - Data Science Central. Accessed November 29, 2015. http://www.datasciencecentral.com/profiles/blogs/random-forests-algorithm

[124] V. Ghosal, “Efficient Face Recognition System using Random Forests” Indian Institute of Technology Kampur. May, 2009. Accessed November 30, 2015. http://www.security.iitk.ac.in/contents/publications/mtech/VidyutGhosal.pdf

[125] Fanelli, Gabriele, Matthias Dantone, Juergen Gall, Andrea Fossati, and Luc Van Gool. "Random Forests for Real Time 3D Face Analysis." *International Journal of Computer Vision Int J Comput Vis* 101, no. 3 (2012): 437-58. doi:10.1007/s11263-012-0549-0.

[126] “Fundamentals of digital image processing." Vernon's Machine Vision. Accessed November 25, 2015. http://homepages.inf.ed.ac.uk/rbf/BOOKS/VERNON/Chap004.pdf

[127] Zivkovic, Zoran, and Ferdinand Van Der Heijden. "Efficient Adaptive Density Estimation per Image Pixel for the Task of Background Subtraction." *Pattern Recognition Letters 27*, no. 7 (2006): 773-80. doi:10.1016/j.patrec.2005.11.005.

[128] Nataraj, Lakshmanan, "A Signal Processing Approach To Malware Analysis”.

[129] Swaroop C H, “A Byte of Python”, Accessed December 5, 2015. http://files.swaroopch.com/python/byte_of_python.pdf

[130] "Welcome to Python.org." Python.org. Accessed December 5, 2015. https://www.python.org/doc/essays/comparisons/

[131] “Python Data Analysis Library." Python Data Analysis Library — Pandas: Python Data Analysis Library. Accessed February 10, 2016. http://pandas.pydata.org/

[132] "Exploit." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Exploit

[133] "Open-source Software." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Open-source_software

[134] "BSD Licenses." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/BSD_licenses

[135] "Multimedia Messaging Service." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Multimedia_Messaging_Service

[136] "Peer-to-peer." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Peer-to-peer

[137] "The Definition of Jailbreak." Dictionary.com. Accessed May 13, 2016. http://www.dictionary.com/browse/jailbreak

[138] "Command and Control (malware)." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Command_and_control_(malware).

[139] "The Definition of URL." Dictionary.com. Accessed May 13, 2016. http://www.dictionary.com/browse/url

[140] "The Definition of USB." Dictionary.com. Accessed May 13, 2016. http://www.dictionary.com/browse/usb

[141] "Man-in-the-middle Attack." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Man-in-the-middle_attack

[142] "POSTER: Role Based Access Control For Android (RBACA)." Accessed May 13, 2016. https://www.acsac.org/2012/program/posters/poster09.pdf

[143] "Address Space Layout Randomization." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Address_space_layout_randomization

[144] "Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis." USENIX. Accessed May 13, 2016. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau

[145] "International Mobile Station Equipment Identity." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/International_Mobile_Station_Equipment_Identity

[146] "Rootkit." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Rootkit

[147] "The Inlined Reference Monitor Approach to Security Policy Enforcement." The Inlined Reference Monitor Approach to Security Policy Enforcement. Accessed May 13, 2016. http://dl.acm.org/citation.cfm?id=997617

[148] "Pixel." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Pixel

[149] "International Mobile Subscriber Identity." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/International_mobile_subscriber_identity

[150] "Transaction Authentication Number." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Transaction_authentication_number

[151] "Android Application Package." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Android_application_package

[152] "Dalvik (software)." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Dalvik_(software)

[153] "Access Point Name." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Access_Point_Name

[154] "Android Runtime." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Android_Runtime

[155] "Wi-Fi." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Wi-Fi

[156] "Internet Bot." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Internet_bot

[157] "Software Development Kit." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Software_development_kit

[158] "Botnet." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Botnet

[159] "Concept of Bits Per Pixel." Www.tutorialspoint.com. Accessed May 13, 2016. http://www.tutorialspoint.com/dip/concept_of_bits_per_pixel.htm

[160] "Out-of-bag Error." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Out-of-bag_error

[161] "A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm." God Your Book Is Great. 2010. Accessed May 13, 2016. https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/

[162] "What Is PNG (Portable Network Graphics)? - Definition from WhatIs.com." SearchSOA. Accessed May 13, 2016. http://searchsoa.techtarget.com/definition/PNG

[163] "Uint8." Matlab. Accessed May 13, 2016. http://www.cs.utah.edu/~germain/PPS/Topics/Matlab/uint8.html

[164] "Epoch (reference Date)." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Epoch_(reference_date)

[165] "Principal Component Analysis." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/Principal_component_analysis

[166] "F1 Score." Wikipedia. Accessed May 13, 2016. https://en.wikipedia.org/wiki/F1_score

[167] Ayoumali. "Face Detection in Transformers." YouTube. 2010. Accessed November 28, 2015. https://www.youtube.com/watch?v=i16ZsCvmOsc

[168] "Gist/Context of a Scene." Home Page. Accessed November 9, 2015. http://ilab.usc.edu/siagian/Research/Gist/Gist.html

[169] Itzamá López Yáñez, Rolando Flores Carapia, Cornelio Yáñez Márquez and Oscar Camacho Nieto, "Automatic detection of cranial fractures in radiological images using a pattern classifier". 2011.

[170] “OpenStax CNX." OpenStax CNX. Accessed November 25, 2015. https://cnx.org/contents/Qp5n91yu@1/Background-Subtraction

[171] "Sklearn.decomposition.PCA." Sklearn.decomposition.PCA — Scikit-learn 0.17.1 Documentation. Accessed November 28, 2015. http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[172]"3.2.4.3.1. Sklearn.ensemble.RandomForestClassifier." 3.2.4.3.1. Sklearn.ensemble.RandomForestClassifier — Scikit-learn 0.17.1 Documentation. Accessed November 29, 2015. http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[173] "Naive Bayes Classifier." Wikipedia. Accessed November 15, 2015. https://en.wikipedia.org/wiki/Naive_Bayes_classifier

Annex I - Code

I.I. Unpacking the Application

extract_F.py

```python
import zipfile
import os
import glob
import sys

folder = 'families'     #new folder created in the location where the command to run this program is executed
os.makedirs(folder)
os.chdir(folder)
path = os.getcwd()
os.chdir('..')
dataset = str(sys.argv[1])  #the parent folder with sub-folders
os.chdir(dataset)
list_fams = os.listdir(os.getcwd())

for i in range(len(list_fams)):
    outpath = path + "/"+ list_fams[i]
    os.makedirs(outpath)
    os.chdir(list_fams[i])
    apk_list = glob.glob('*.apk') # Getting only 'apk' files in a folder
    for j in range(len(apk_list)):
        fh = open(apk_list[j], 'rb')
        z = zipfile.ZipFile(fh)
        name = str(apk_list[j])
        name = name[:-4]
        outpath = path+"/"+ list_fams[i]+"/"+name   #store the extracted files in the folder "families" inside its corresponding family
        os.makedirs(outpath)
        z.extractall(outpath)   #extract all the files inside an APK
        fh.close()
    os.chdir('..')

```

FIGURE 40. EXTRACT_F.PY

extract_GM.py

```python
import zipfile
import os
import glob
import sys

folder = 'good_mal' #new folder created in the location where the command to run this program is executed
os.makedirs(folder)
os.chdir(folder)
path = os.getcwd()
os.chdir('..')
dataset = str(sys.argv[1])  #the parent folder with sub-folders
os.chdir(dataset)
list_fams = os.listdir(os.getcwd())
for i in range(len(list_fams)):
    outpath = path + "/"+ list_fams[i]
    os.makedirs(outpath)
    os.chdir(list_fams[i])
    folder_name = "goodwares"
    listi = str(list_fams[i])
    if folder_name == listi:
        zip_list = glob.glob('*.benign')    #Getting only 'bening' files in a folder
        zip_list.extend(glob.glob('*.apk'))     #Adding only 'apk' files to the previous list
    else:
        zip_list = glob.glob('*.malware')   #Getting only 'malware' files in a folder


    for j in range(len(zip_list)):
        fh = open(zip_list[j], 'rb')
        z = zipfile.ZipFile(fh)
        count = 0
        for zinfo in z.infolist():
                is_encrypted = zinfo.flag_bits & 0x1    #check if the files are encrypted
            if is_encrypted:
                print '%s is encrypted!' % zinfo.filename
                count = 1
            else:
                print '%s is not encrypted!' % zinfo.filename

        if count == 0:
            name = str(zip_list[j])
            name = name[:-7]
            outpath = path+"/"+ list_fams[i]+"/"+name   #store the extracted files in the folder "good_mal" inside its corresponding family(goodware/malware)
            os.makedirs(outpath)
            z.extractall(outpath)   #extract all the files inside an APK
            fh.close()
    os.chdir('..')

```

FIGURE 41. EXTRACT_GM.PY

I.II. Transform classes.dex into PNG Image

```
convert2image.py

1   #!/usr/bin/env python
2   # -*- coding: utf-8 -*-
3   from PIL import Image
4   import numpy,scipy.misc, os, array
5   import glob
6   import sys
7   #STEP 0. Convert the malware if it is in binary format to an image
8
9   folder = str(sys.argv[1])
10  os.chdir(folder) # name of the dataset to be transformed into images: it can be 'families' or 'good_mal'
11          # the parent folder with sub-folders
12  list_fams = os.listdir(os.getcwd())
13
14  for i in range(len(list_fams)):
15      os.chdir(list_fams[i])
16      list_fams2 = os.listdir(os.getcwd())
17      for j in range(len(list_fams2)):
18          os.chdir(list_fams2[j])
19          filename = 'classes.dex'
20          if os.path.exists(filename):
21              f = open(filename,'rb')
22              ln = os.path.getsize(filename)    # length of file in bytes
23
24              if ln < 10000:
25                  width = 32
26              elif ln >= 10000 and ln<30000:
27                  width = 64
28              elif ln >= 30000 and ln<60000:
29                  width = 128
30              elif ln >= 60000 and ln<100000:
31                  width = 256
32              elif ln >= 100000 and ln<200000:
33                  width = 384
34              elif ln >= 200000 and ln<500000:
35                  width = 512
36              elif ln >= 500000 and ln<1000000:
37                  width = 768
38              else:
39                  width = 1024
40
41
42              rem = ln%width
43              array1 = array.array("B") # uint8 array (0-255)
44              array1.fromfile(f,ln-rem) #copy the content of the file
45              f.close()
46              array_img = numpy.reshape(array1,(len(array1)/width,width)) #reshape the array to have the size of the final image (length/width x width)
47              array_img = numpy.uint8(array_img)
48              malware = list_fams2[j] + ".png" #get the name of the apk file and use the same name to save the image
49              os.chdir('..')
50              scipy.misc.imsave(malware ,array_img) # save the image as png
51          else:
52              print "classes.dex does not exist"
53              os.chdir('..')
54      os.chdir('..')
55
```

FIGURE 42. CONVERT2IMAGE.PY

- Some lines of the previous code would be affected in case of converting directly an app to PNG images.

```
16      apk_list = glob.glob('*.apk')
17      for j in range(len(apk_list)):
18          filename = apk_list[j]
```

FIGURE 43. MALGENOME APK TO PNG

```
16      folder_name = "goodwares"
17      listi = str(list_fams[i])
18      if folder_name == listi:
19          apk_list = glob.glob('*.benign')    #Getting only 'bening' files in a folder
20          apk_list.extend(glob.glob('*.apk'))     #Adding only 'apk' files to the previous list
21      else:
22          apk_list = glob.glob('*.malware')   #Getting only 'malware' files in a folder
23      for j in range(len(apk_list)):
24          filename = apk_list[j]
```

FIGURE 44. M0DROID APK TO PNG

I.III. Subtraction Classification

sub_classif.py

```
from pandas_confusion import ConfusionMatrix
import matplotlib.pyplot as plt
import numpy as np
import os, glob
from PIL import Image,ImageChops
import sys

folder = str(sys.argv[1])    #the parent folder with sub-folders obtained as an attribute from the command line
os.chdir(folder)
list_fams = os.listdir(os.getcwd())
training = [0 for t in range(len(list_fams))]   #array that contains the training samples
test = [0 for t in range(len(list_fams))]   #array that contains the testing samples
testing_images = [0 for t in range(len(list_fams))] #array that contains the number of testing samples for each family
for i in range(len(list_fams)):
    os.chdir(list_fams[i])
    img_list = glob.glob('*.png')   #get png files stored in each sub-folder (class folder)
    np.random.shuffle(img_list) #randomly shuffle the samples of each family to get different samples for each test
    #calculate number of images selected for training and testing
    len_training = int(len(img_list)*0.8)
    training[i], test[i] = img_list[:len_training], img_list[len_training:]
    testing_images[i] = len(img_list) - len_training    #save the total number of testing images of each family
    os.chdir('..')

len_testing = sum(testing_images)
accuracy = 0
y_actu = [0 for t in range(len_testing)]
y_pred = [0 for t in range(len_testing)]
count = 0
for i in range(len(test)):
    for j in range(len(test[i])):
        os.chdir(list_fams[i])
        im1 = Image.open(test[i][j])    #open testing image of each family
        width1, height1 = im1.size
        min_family = [0 for t in range(len(training))]
        os.chdir('..')
        for k in range(len(training)):
            os.chdir(list_fams[k])
            total=[0 for t in range(len(training[k]))]
            equal = 0
            for w in range(len(training[k])):
                im2 = Image.open(training[k][w])    #open training image of each family
                width2, height2 = im2.size

                #resize the images to be the same size
                if height2 == height1:
                    if width1 == width2:
                        print "no resize needed"
                    elif width2<width1:
                        im1.resize((width2,height2),Image.ANTIALIAS);

                    elif width1<width2:
                        im2.resize((width1,height1),Image.ANTIALIAS);

                    else:
                        print "error"
                elif height2 < height1 :
                    if width1 == width2:
                        im1.resize((width2,height2),Image.ANTIALIAS);

                    elif width2<width1:
                        im1.resize((width2,height2),Image.ANTIALIAS);

                    elif width1<width2:
                        im1.resize((width1,height2),Image.ANTIALIAS);
                        im2.resize((width1,height2),Image.ANTIALIAS);

                    else:
                        print "error"

                elif height1 < height2:
                    if width1 == width2:
                        im2.resize((width1,height1),Image.ANTIALIAS);

                    elif width2<width1:
                        im2.resize((width2,height1),Image.ANTIALIAS);
                        im1.resize((width2,height1),Image.ANTIALIAS);

                    elif width1<width2:
                        im2.resize((width1,height1),Image.ANTIALIAS);

                    else:
                        print "error"
                else:
                    print "error"
                diff = ImageChops.difference(im2, im1)  #difference between training image and unclassified testing image
                color = 0
                pix = diff.load()
                width, height = diff.size
                for x in range(width):
                    for y in range(height):
                        color= pix[x,y] + color #sum the color of each pixel
                size = width * height
                total[w] = color/size   #quantity of color in the resulting image
                if total[w] == 0:   #if the color is 0 (completely black image) ==> Stop subtracting more training images with testing image
                    equal = 1
                    break
            if equal == 0:
                min_family[k] = min(total)  #get the minimum color value of all the images subtracted of this current family
                print "Avg quantity of color: " + str(min_family[k]) + " of family " + str(list_fams[k]) + " with malware unclassified " +  str(test[i][j])
                os.chdir('..')
            else:
                min_family[k] = -1;     #there was a black subtracted image so the unclassified malware belongs to this current family
                print "There is a match between family " + str(list_fams[k]) + " with malware unclassified " +  str(test[i][j])
                os.chdir('..')
                break
        minimum = min(min_family)   #obtain the minimum color value of all the families
        ind = min_family.index(minimum)     #get the index of the family name of the selected color value
        print "malware " + str(test[i][j]) + " probably belongs to " + str(list_fams[ind])
        y_actu[count] = list_fams[i]    #get the actual family name of the unclassified malware
        y_pred[count] = list_fams[ind]  #get the predicted family name of the unclassified malware
        count = count +1
        if i==ind:  #if they point to the same family
            print "well-classified"
        else:
            print "wrong-classified"

#compute and print confusion matrix
confusion_matrix = ConfusionMatrix(y_actu, y_pred)
confusion_matrix.print_stats()
confusion_matrix.plot(normalized=True)
plt.show()
```

FIGURE 45. SUB_CLASSIF.PY

I.IV. Extracting Features and Machine Learning Classification

- Step 1. Compute label matrix

```
classif.py
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from pandas_confusion import ConfusionMatrix
5  import numpy,scipy, os, array, Image
6  import glob,random
7  from sklearn.cross_validation import StratifiedKFold
8  from sklearn.utils import shuffle
9  import matplotlib.pyplot as plt
10 import pickle as cPickle
11 import time
12 import sys
13
14 #FEATURE EXTRACTION
15 from sklearn.feature_extraction import image
16 from skimage.feature import daisy
17
18 #FEATURES SELECTION
19 from sklearn.decomposition import PCA
20
21 #CLASSIFIERS
22 import leargist
23 from sklearn.neighbors import KNeighborsClassifier
24 from sklearn.ensemble import RandomForestClassifier
25 from sklearn.naive_bayes import GaussianNB
26 from sklearn.tree import DecisionTreeClassifier
27
28
29
30
31 #COMPUTE THE LABEL MATRIX (y)
32
33 folder = str(sys.argv[1]) #the parent folder with sub-folders obtained as an attribute from the command line
34 os.chdir(folder)
35
36 list_fams = os.listdir(os.getcwd())      #vector of strings with family names
37 y1 = []
38 no_imgs = [0 for t in range(len(list_fams))]     #number of samples per family
39 for i in range(len(list_fams)):      #get the number of samples per family
40     os.chdir(list_fams[i])
41     len1 = len(glob.glob('*.png'))      #get the number of png files stored in each sub-folder (class folder)
42     no_imgs[i] = len1    #save the total number of images of each family
43     for j in range(len1):
44         y1.append(list_fams[i])      #save the family name (label) of each image
45     os.chdir('..')
46
47 y = numpy.asarray(y1)    #array that contains the label for each image
```

FIGURE 46. CLASSIF.PY (PART 1)

- Step 2. Compute features matrix

```
47  #COMPUTE THE FEATURES MATRIX (X)
48
49  #IMG+HISTO
50  #hist = numpy.zeros(((sum(no_imgs)),400))
51  #dai = numpy.zeros(((sum(no_imgs)),400))
52  #gis = numpy.zeros(((sum(no_imgs)),400))
53  n_des = 400 #number of descriptors retrieved from an image descriptor
54  X = numpy.zeros((sum(no_imgs),n_des)) # Feature Matrix with two matrixes inside: one for no_images and another with the number of descriptors used
55  tic = time.time()   #extracting features time (initial time)
56  for i in range(len(list_fams)):
57      os.chdir(list_fams[i])
58      img_list = glob.glob('*.png')   #get png files stored in each sub-folder (class folder)r
59      for j in range(len(img_list)):
60
61          #GIST descriptor (uncomment if you want to use this image descriptor)
62          #im = Image.open(img_list[j])
63          #im1 = im.resize((35,35),Image.ANTIALIAS);   #resize image for faster computation
64              #des = leargist.color_gist(im1)         #apply GIST descriptor to the image
65          #X[cnt]=des[0:n_des]    #store a specific number of descriptors in X matrix
66
67          #HISTOGRAM (uncomment if you want to use this descriptor)
68          #im = Image.open(img_list[j])
69          #im1 = im.resize((35,35),Image.ANTIALIAS);  #resize image for faster computation
70          #X[cnt],bins = numpy.histogram(numpy.ravel(im1),n_des,[0,n_des])    #store a specific number of histogram descriptors in X matrix
71
72          #DAISY (comment if you want to use another descriptor)
73          im = Image.open(img_list[j])
74          im1 = im.resize((35,35),Image.ANTIALIAS)    #resize image for faster computation
75          des, descs_img = daisy(im1) #appy daisy descriptor to the image
76          n = n_des/2
77          d0=descs_img[0][0:n]
78          d1=descs_img[1][0:n]
79          X[cnt] = numpy.hstack((d0,d1))  #stack the two arrays in sequence horizontally (column wise) to store the features values for each image
80
81          #image to graph (uncomment if you want to use this descriptor)
82          #im = Image.open(img_list[j])
83          #im1 = im.resize((35,35),Image.ANTIALIAS);  #resize image for faster computation
84              #graph = image.img_to_graph(im1)    #apply img_to_graph descriptor
85          #graph.data = numpy.exp(-graph.data/graph.data.std())    #obtain the data from the graph retrieved
86          #X[cnt]=graph.data[0:n_des] #store a specific number of histogram descriptors in X matrix
87
88          #HISTOGRAM+DAISY (uncomment if you want to use this descriptor)
89          #im = Image.open(img_list[j])
90          #im1 = im.resize((35,35),Image.ANTIALIAS);
91          #n = n_des/4
92          #n2 = n_des/2
93          #des, descs_img = daisy(im1)
94          #d0=descs_img[0][0:n]
95          #d1=descs_img[1][0:n]
96          #dai[cnt] = numpy.hstack((d0,d1))
97          #hist[cnt],bins = numpy.histogram(numpy.ravel(im1),n2,[0,n2])
98
99          #DAISY + GIST (uncomment if you want to use this descriptor)
100         #im = Image.open(img_list[j])
101         #im1 = im.resize((35,35),Image.ANTIALIAS);
102         #des, descs_img = daisy(im1)
103         #n = n_des/4
104         #n2 = n_des/2
105         #d0=descs_img[0][0:n]
106         #d1=descs_img[1][0:n]
107         #dai[cnt] = numpy.hstack((d0,d1))
108         #des = leargist.color_gist(im1)
109         #gis[cnt]=des[0:n2]
110
111
112             cnt = cnt + 1
113     os.chdir('..')
114
115 #X = numpy.hstack((hist,dai)) #HISTOGRAN+DAISY
116 #X = numpy.hstack((dai,gis)) #DAISY+GIST
117 toc = time.time()   #extracting features time (passed time)
118 print "computing X time = ", toc-tic    #final extracting features time (passed time - initial time)
```

FIGURE 47. CLASSIF.PY (PART 2)

- Step 3. Supervised Classification

```
125  #SUPERVISED CLASSIFICATION
126
127  #Divide the data into folds to obtaing train and testing sets
128  n_samples, n_features = X.shape
129  p = range(n_samples)     #obtain an index array from 0-n_samples
130  random.seed(random.random())
131  random.shuffle(p)        #shuffle the index array
132  X, y = X[p], y[p]        #shuffle the X and y matrices to avoid getting always the same values
133  kfold = 2            #number of folds
134  skf = StratifiedKFold(y,kfold)  #cross-validation iterator that provides train and testing indices
135  skfind = [None]*len(skf)    #indices
136  cnt=0
137  for train_index in skf:
138    skfind[cnt] = train_index #skfind[i][0] -> train indices, skfind[i][1] -> test indices
139    cnt = cnt + 1
140
141
142  #Supervised Classification with 2-fold Cross Validation
143  n_neighbors = 1;
144  y_actu = []
145  y_pred = []
146  #2-fold Cross Validation
147  for i in range(kfold):
148      train_indices = skfind[i][0]
149      test_indices = skfind[i][1]
150      clf = []
151      clf = KNeighborsClassifier(n_neighbors,weights='distance')  #KNN classification (comment if you want to use another classifier)
152      #clf = GaussianNB()        #Gaussian Naive Bayes classification (uncomment if you want to use this classifier)
153      #clf = DecisionTreeClassifier(random_state=0)  #Decision Tree classification (uncomment if you want to use this classifier)
154      #clf = RandomForestClassifier(n_estimators=10)  #Random Tree classification (uncomment if you want to use this classifier)
155      pca = PCA(n_components=25)  #PCA features selection
156      X_train = X[train_indices]
157      y_train = y[train_indices]
158      X_test = X[test_indices]
159      y_test = y[test_indices]
160      pca.fit(X_train)
161      X_train_pca = pca.transform(X_train)
162      X_test_pca = pca.transform(X_test)
163
164
165
166    # Training
167      tic = time.time()    #training time (initial time)
168      clf.fit(X_train_pca, y_train)       #do the training to have a final model to predict and classify new values with PCA
169      #clf.fit(X_train, y_train)      #do the training to have a final model to predict and classify new values
170      toc = time.time()    #training time (passed time)
171      print "training time= ", toc-tic     #final training time (passed time - initial time)
172
173    # Testing
174      y_predict = []
175      tic = time.time()       #testing time (initial time)
176      y_predict = clf.predict(X_test_pca)      #make the prediction with the testing values with PCA
177      #y_predict = clf.predict(X_test)     #make the prediction with the testing values
178      toc = time.time()       #testing time (passed time)
179      print "testing time = ", toc-tic   #calculate testing time (passed time - initial time)
180      y_actu.extend(y_test)
181      y_pred.extend(y_predict)
182
183
184  #COMPUTE AND PRINT THE CONFUSION MATRIX
185  confusion_matrix = ConfusionMatrix(y_actu, y_pred)
186  confusion_matrix.print_stats()
187  confusion_matrix.plot(normalized=True)
188  plt.show()
189  #plt.savefig('confusion_matrix.png')
```

FIGURE 48. CLASSIF.PY (PART 3)

Annex II - Confusion Matrices Images

II. I. Subtraction Classification

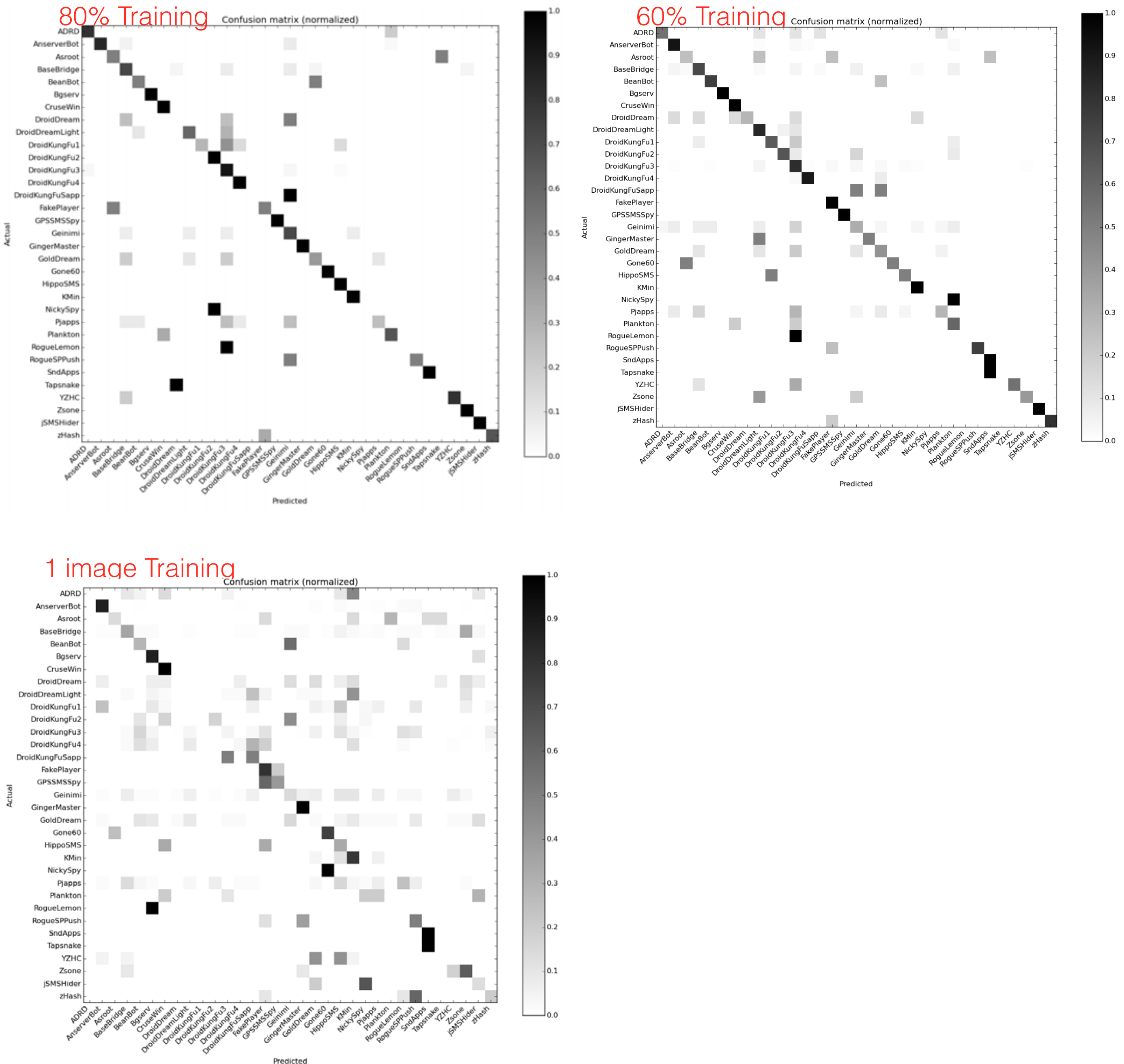


FIGURE 49. SUBTRACTION CLASSIFICATION CONFUSION MATRICES

II. II. Extracting Features + Classification

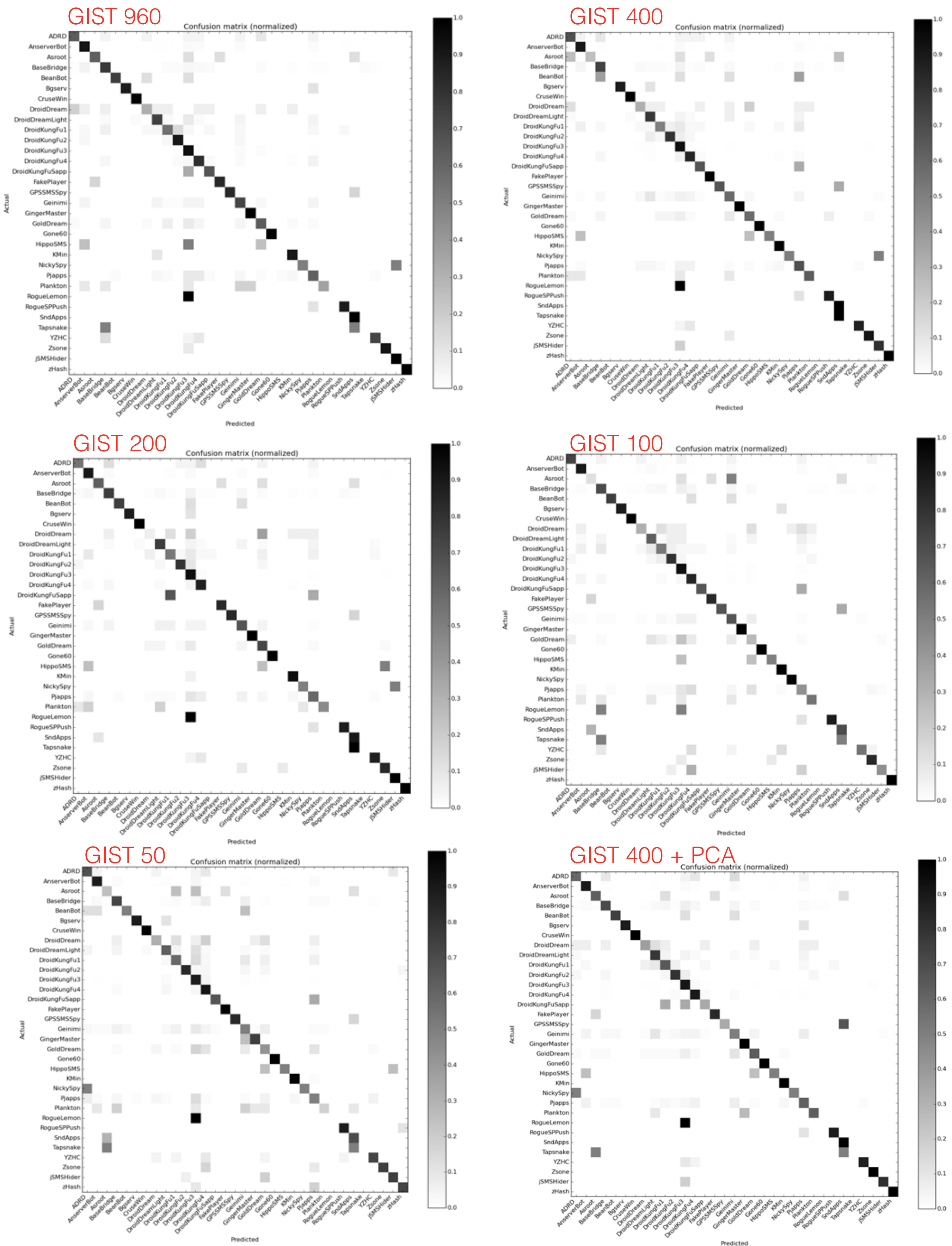


FIGURE 50. GIST + KNN

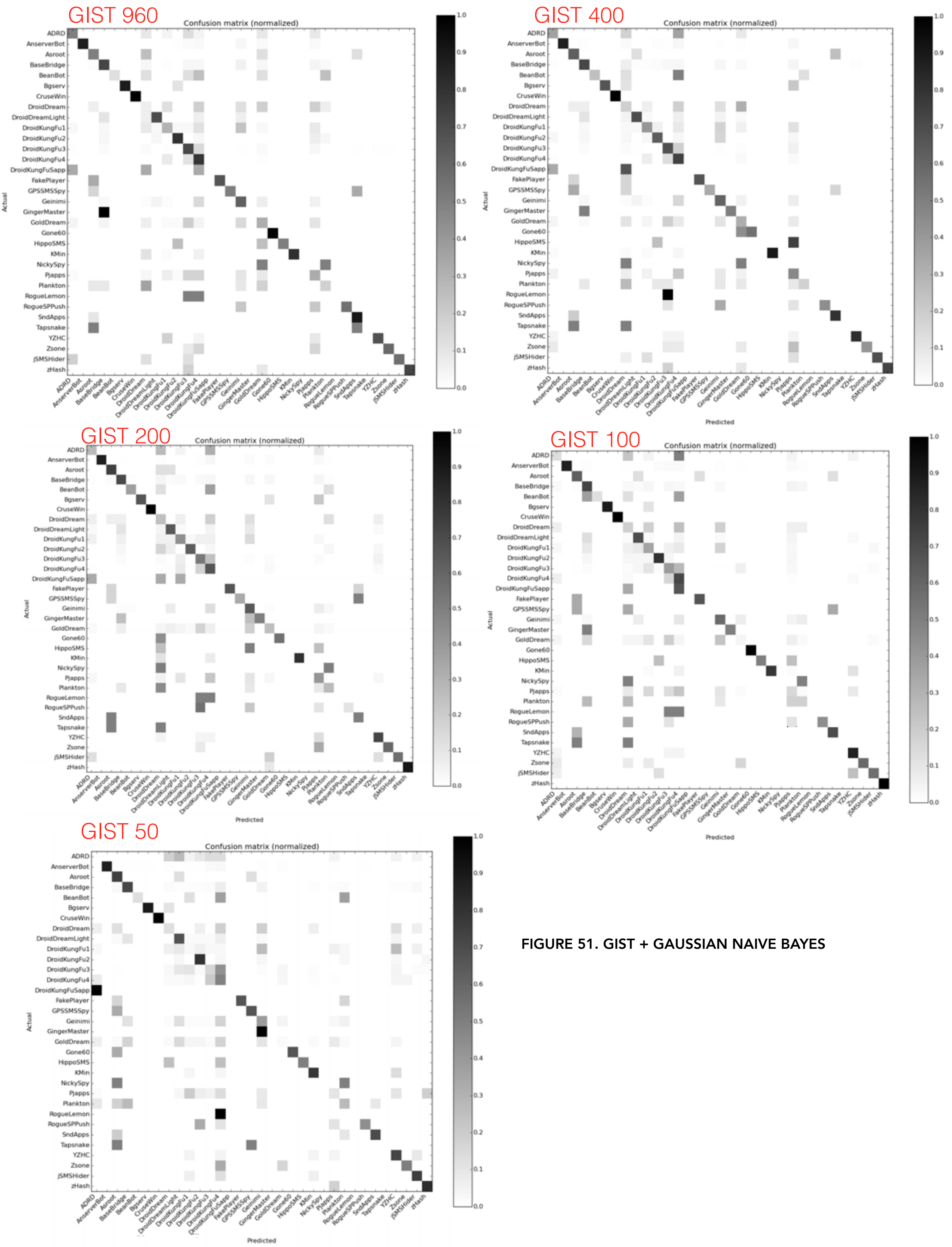
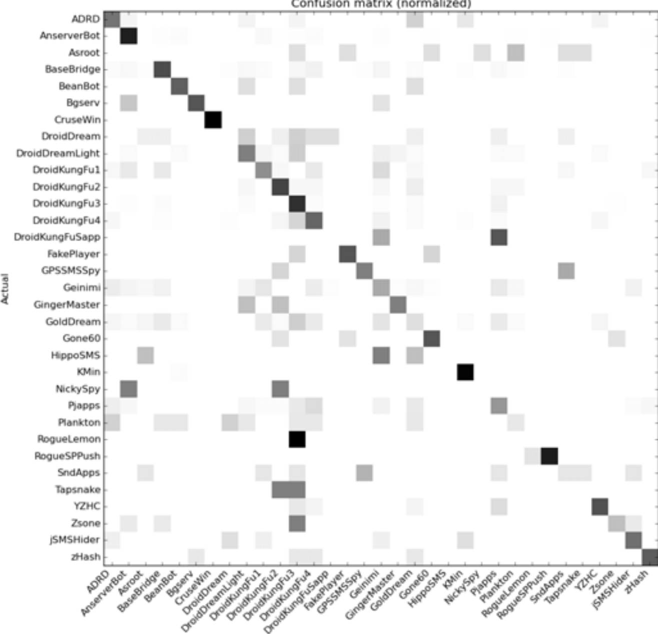
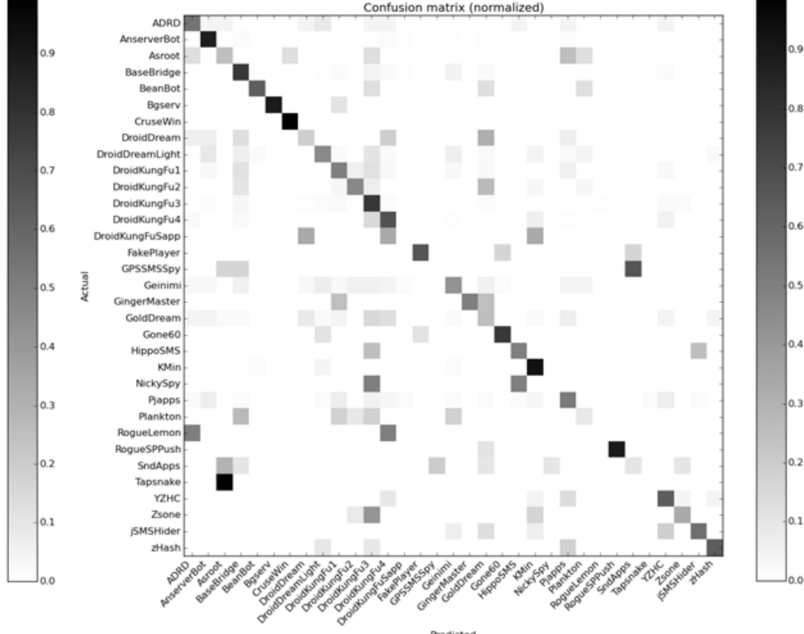
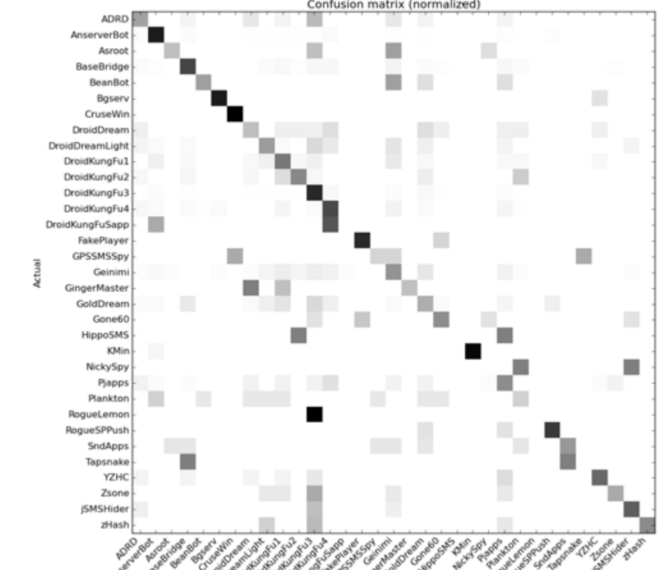
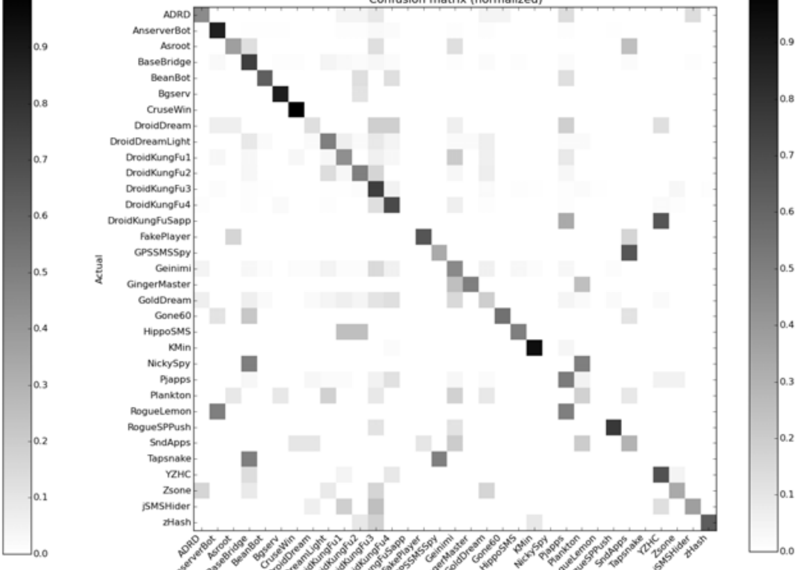


FIGURE 51. GIST + GAUSSIAN NAIVE BAYES









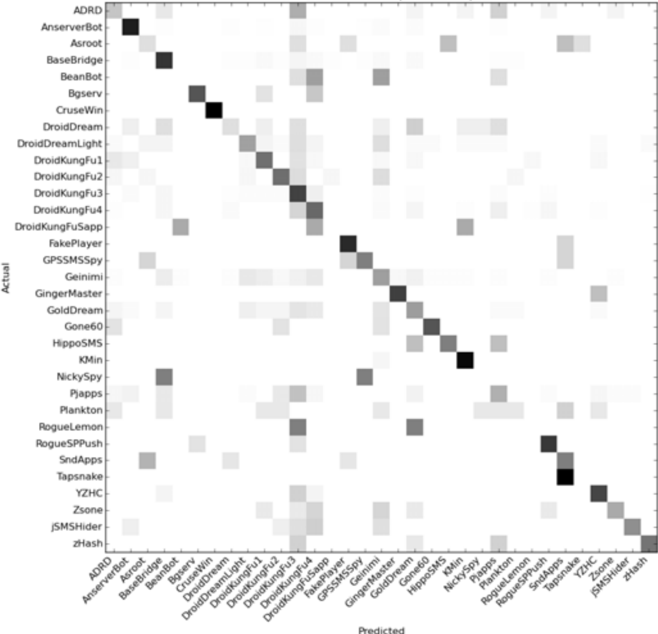


FIGURE 52. GIST + DECISION TREE

GIST 960

Confusion matrix (normalized)

GIST 400

Confusion matrix (normalized)

GIST 200

Confusion matrix (normalized)

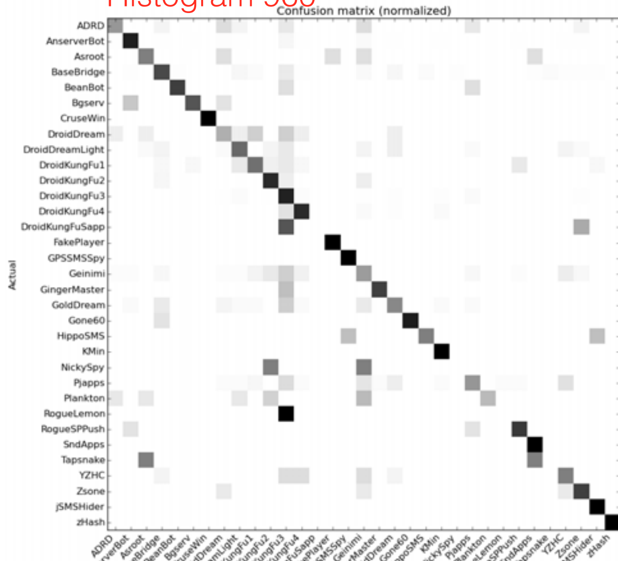
GIST 100

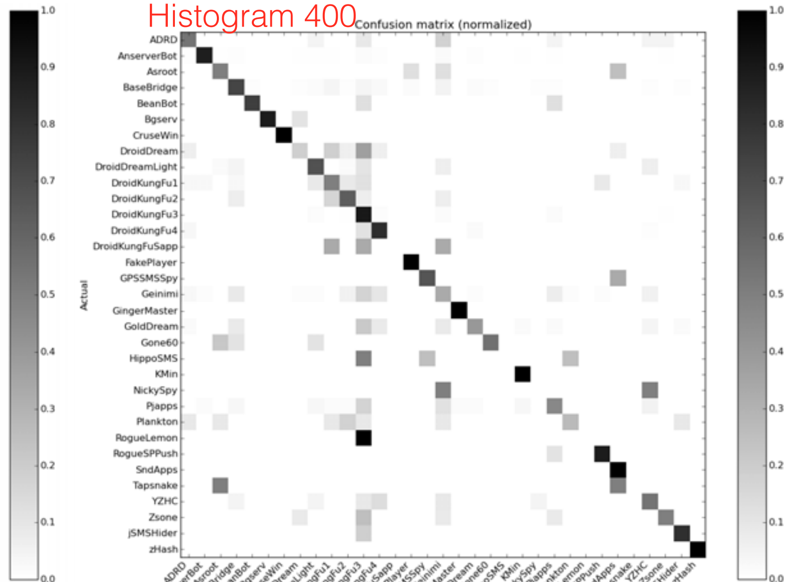
Confusion matrix (normalized)

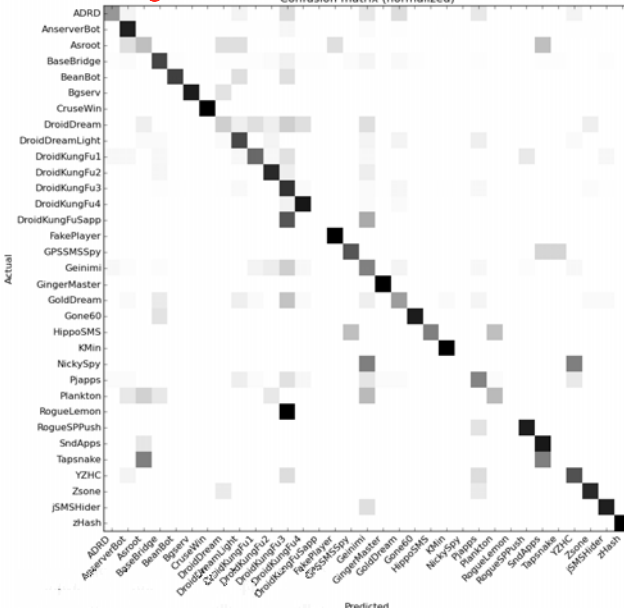
GIST 50

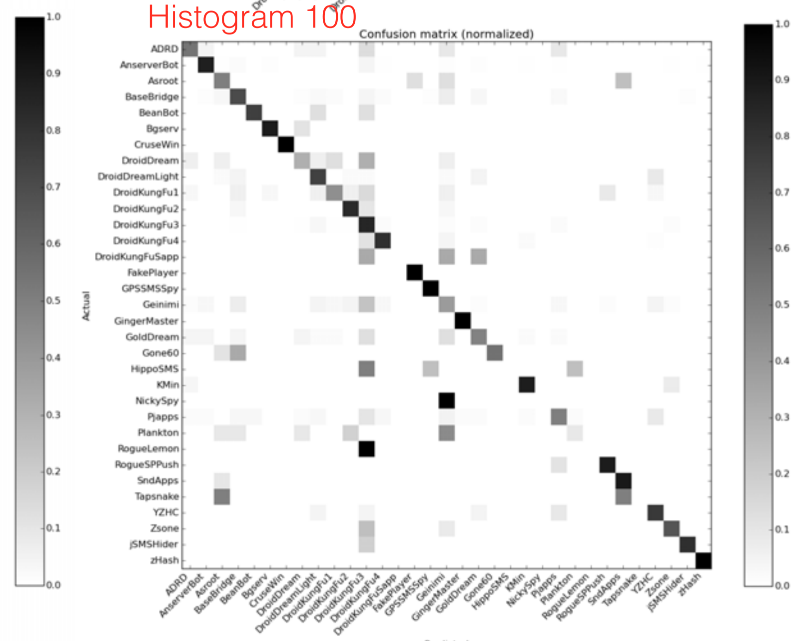
Confusion matrix (normalized)

FIGURE 53. GIST + RANDOM FOREST









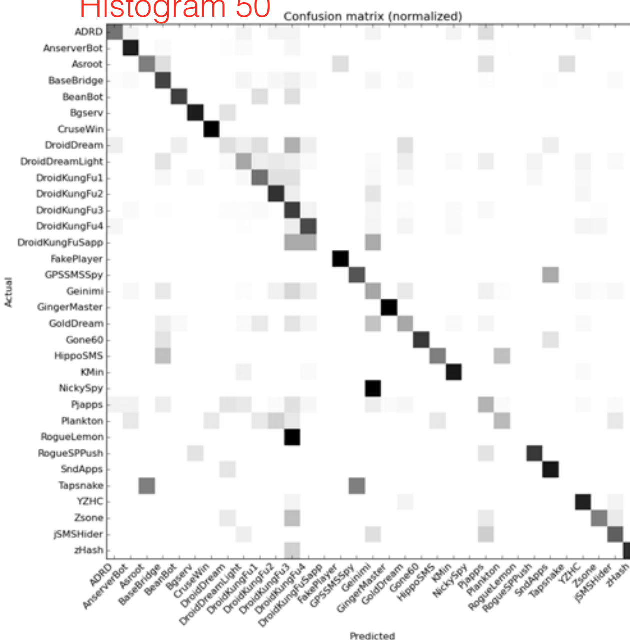
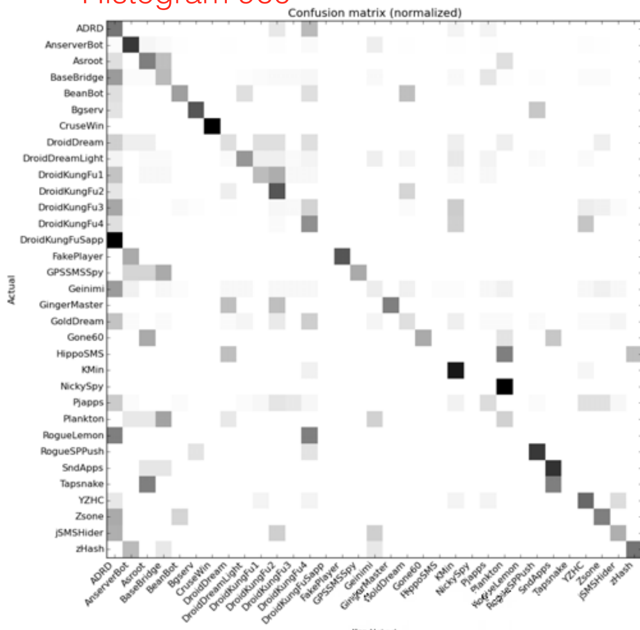
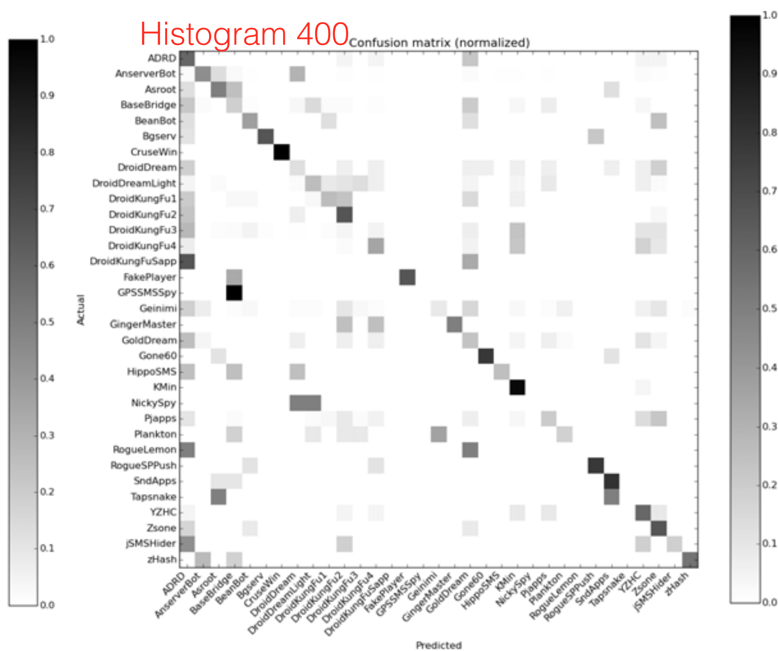
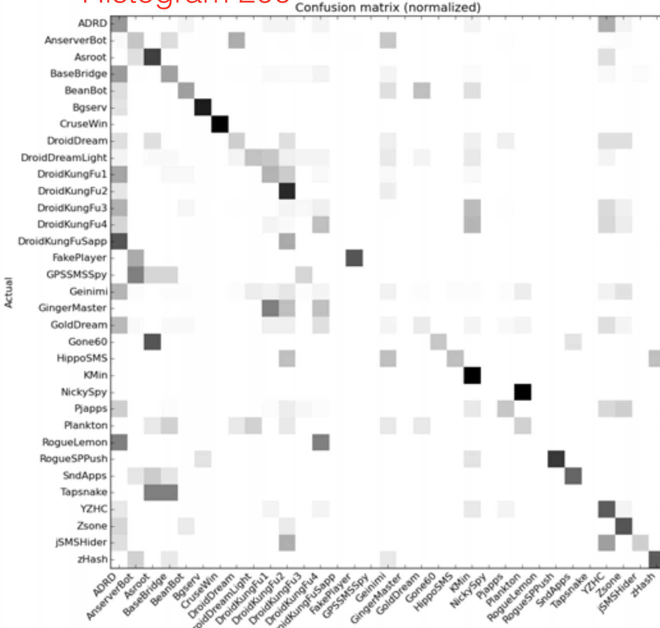
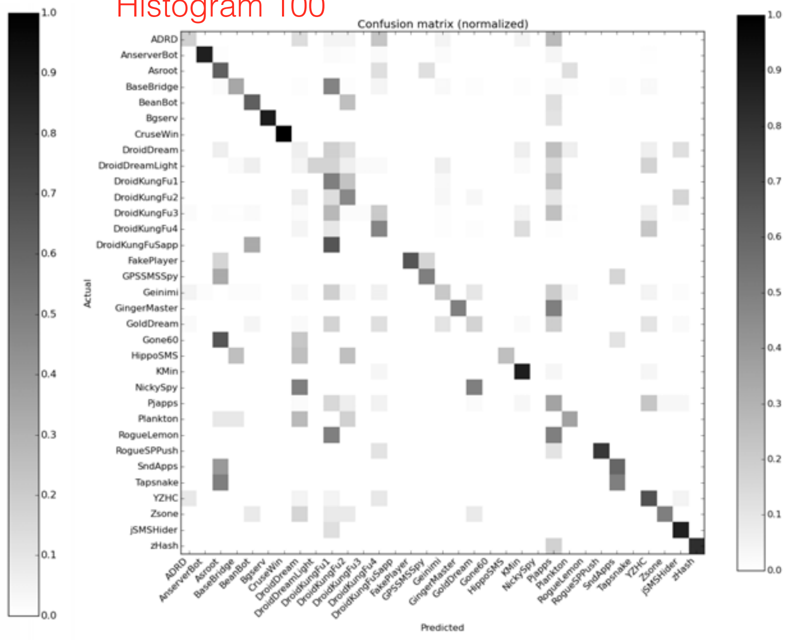


FIGURE 54. HISTOGRAM + KNN









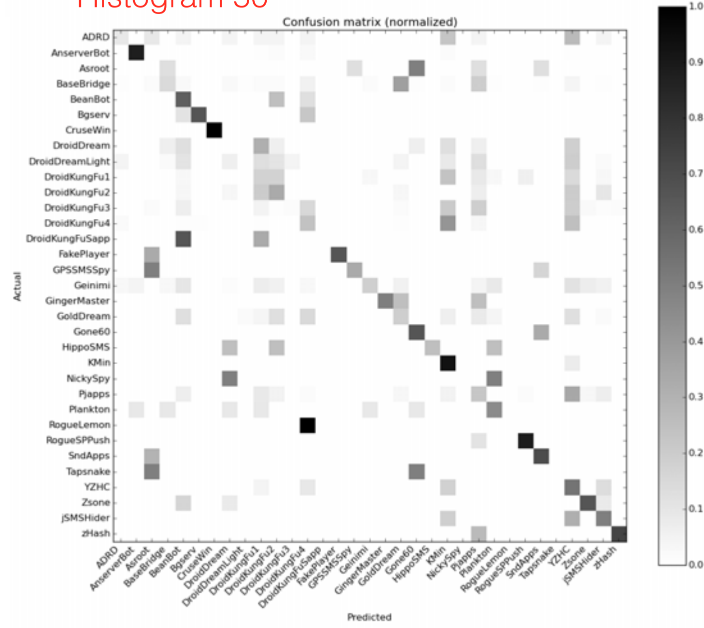
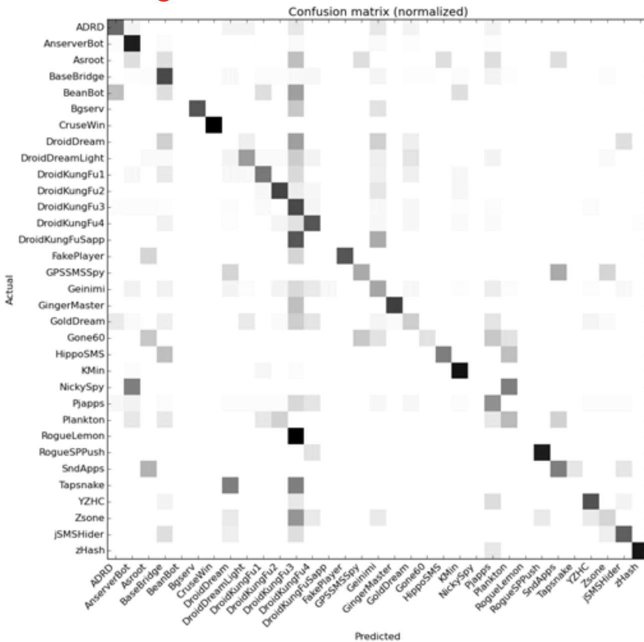
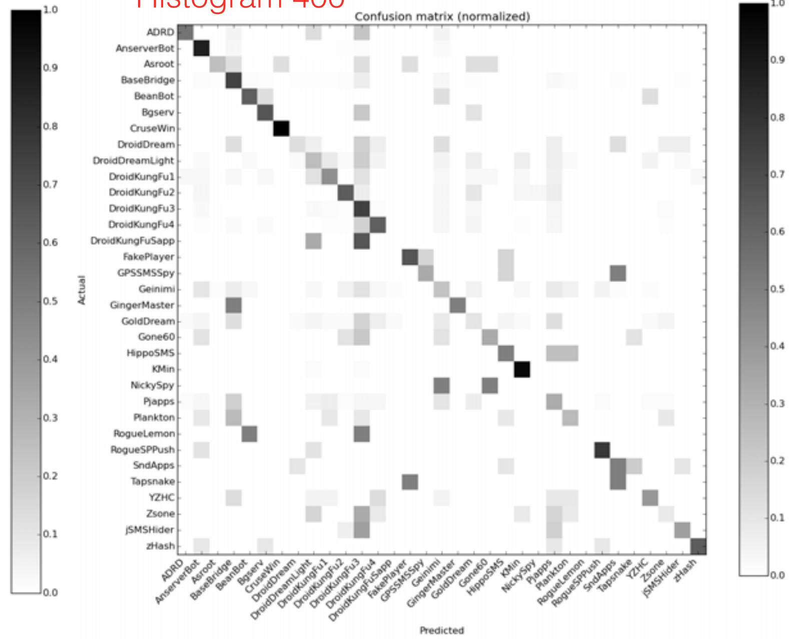


FIGURE 55. HISTOGRAM + GAUSSIAN NAIVE BAYES

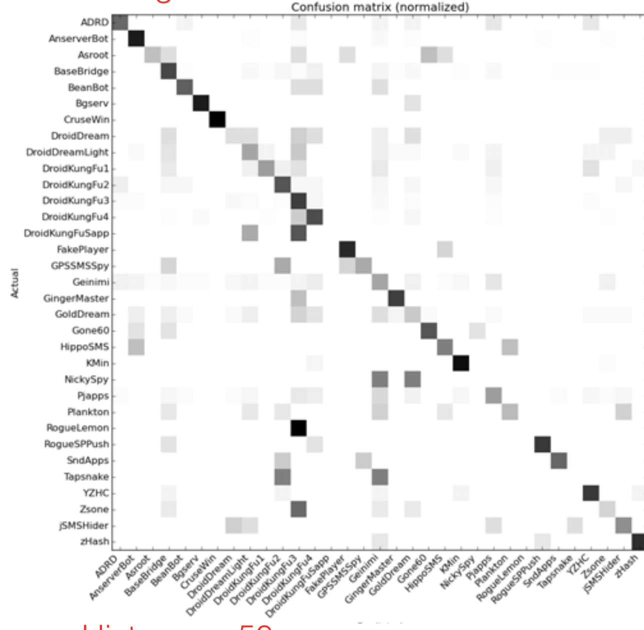
Histogram 960



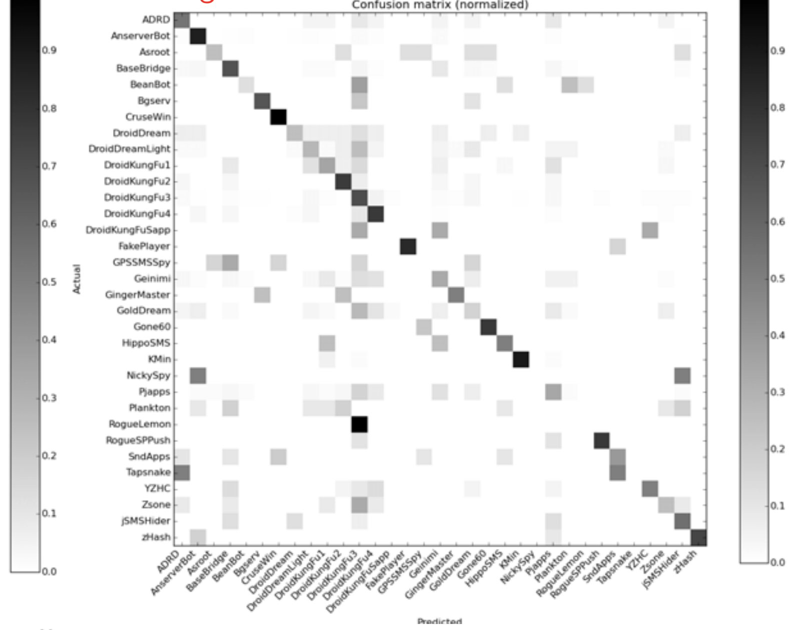
Histogram 400



Histogram 200



Histogram 100



Histogram 50

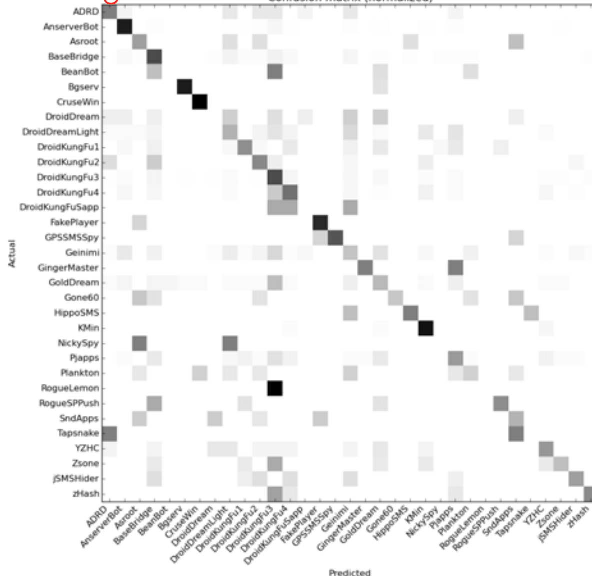
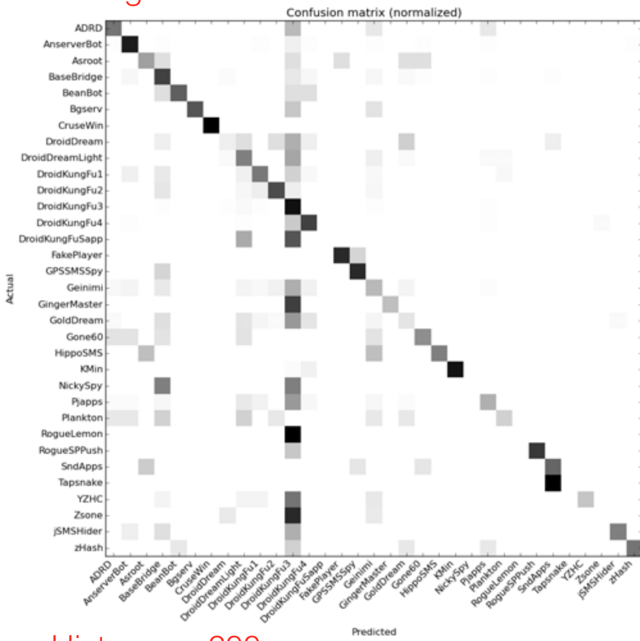
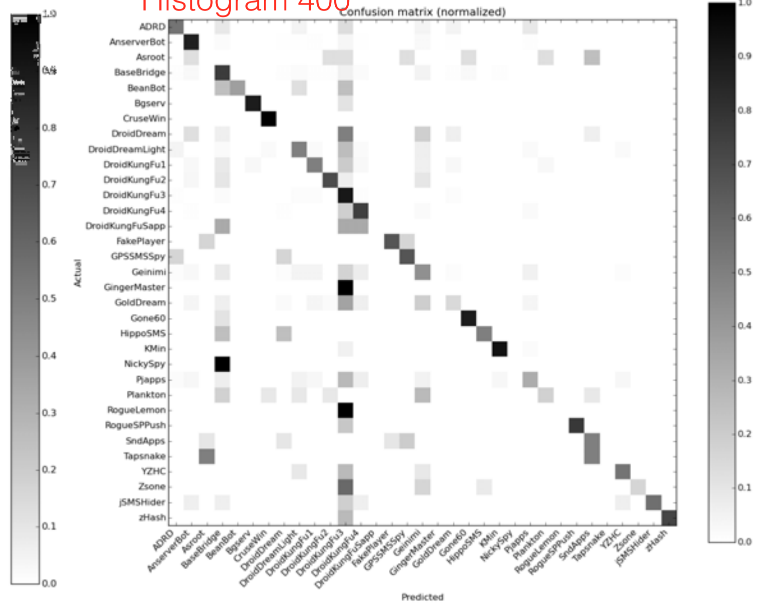
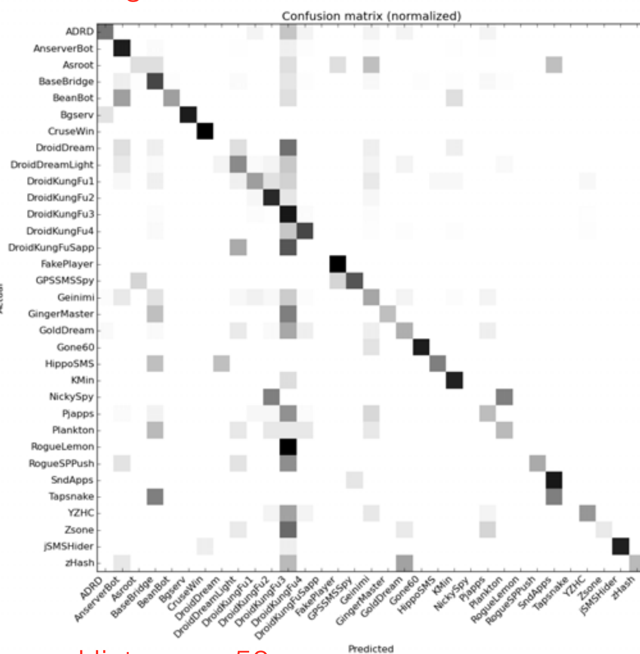
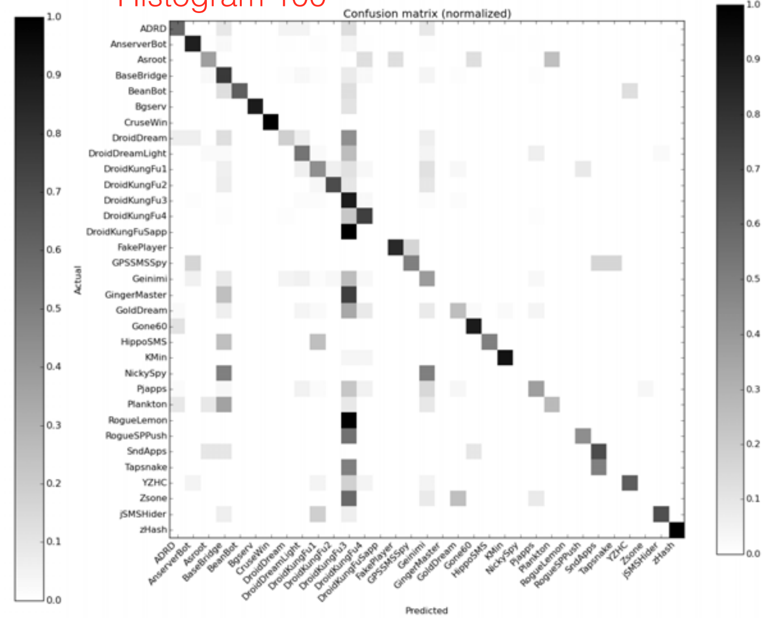


FIGURE 56. HISTOGRAM + DECISION TREE









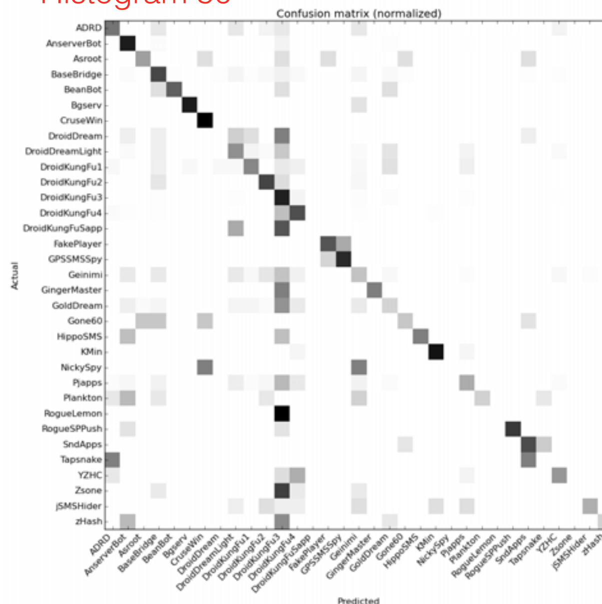
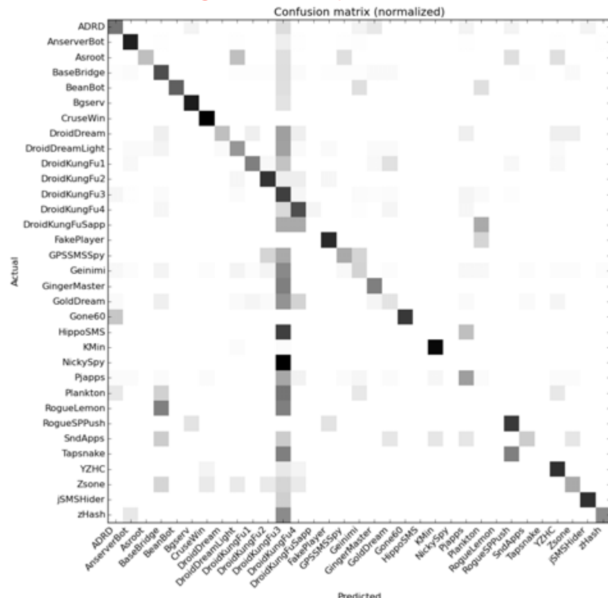
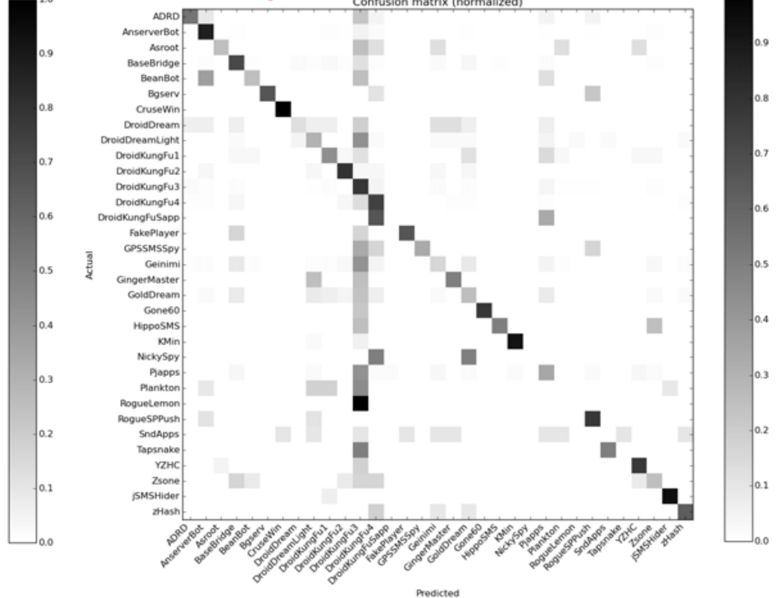
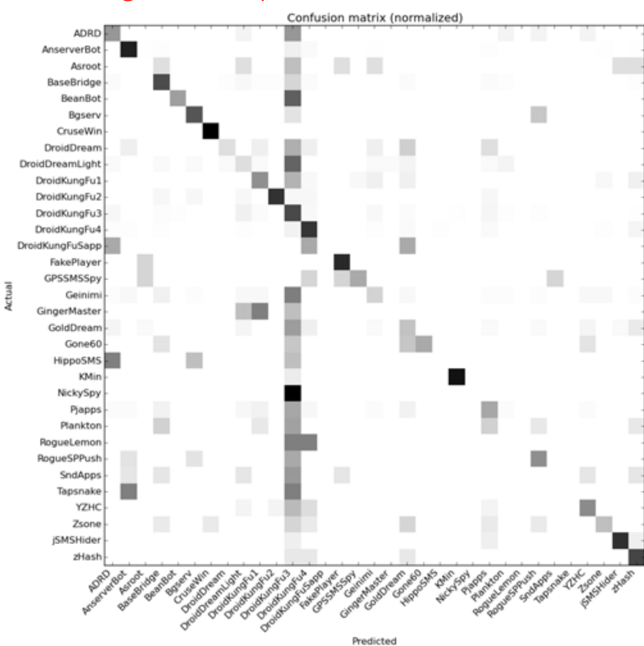
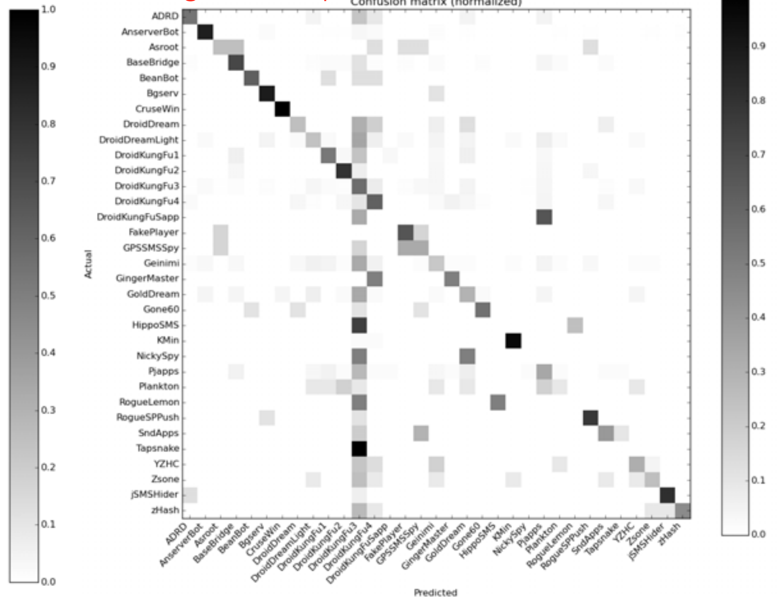


FIGURE 57. HISTOGRAM + RANDOM FOREST









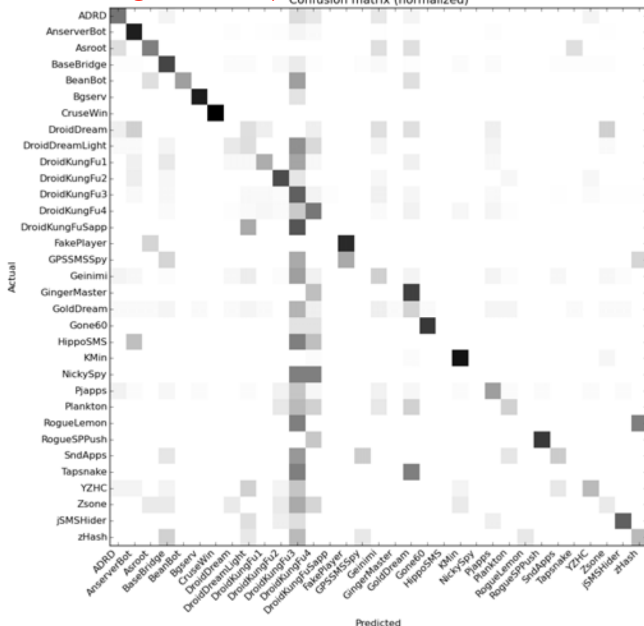


FIGURE 58. IMAGE TO GRAPH + KNN

Image To Graph 960

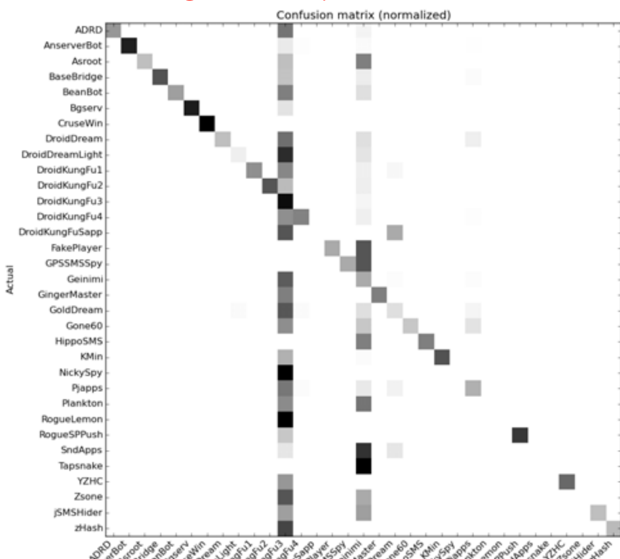


Image To Graph 400

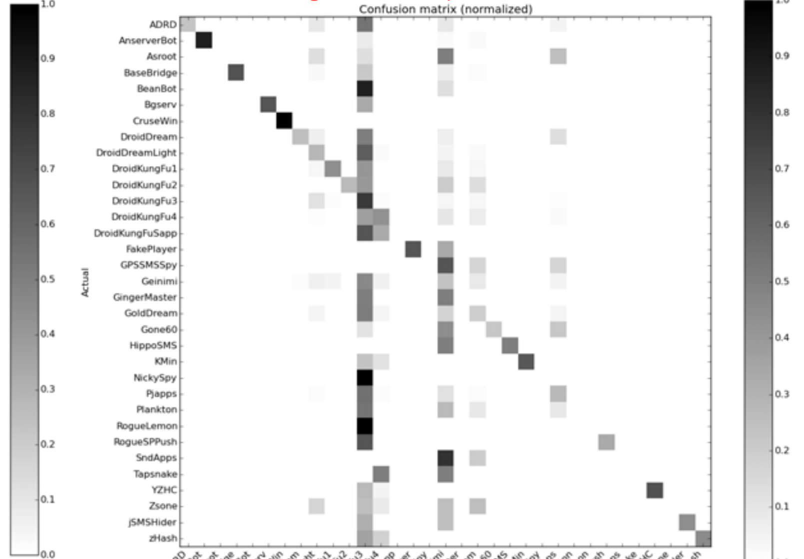


Image To Graph 200

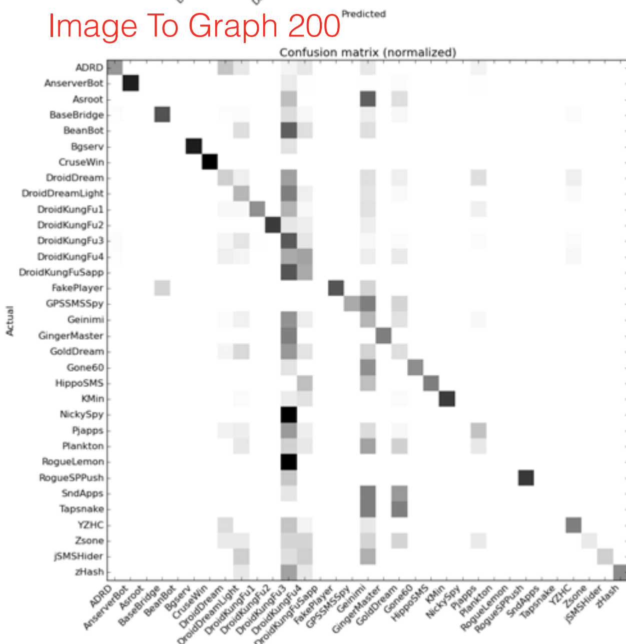


Image To Graph 100

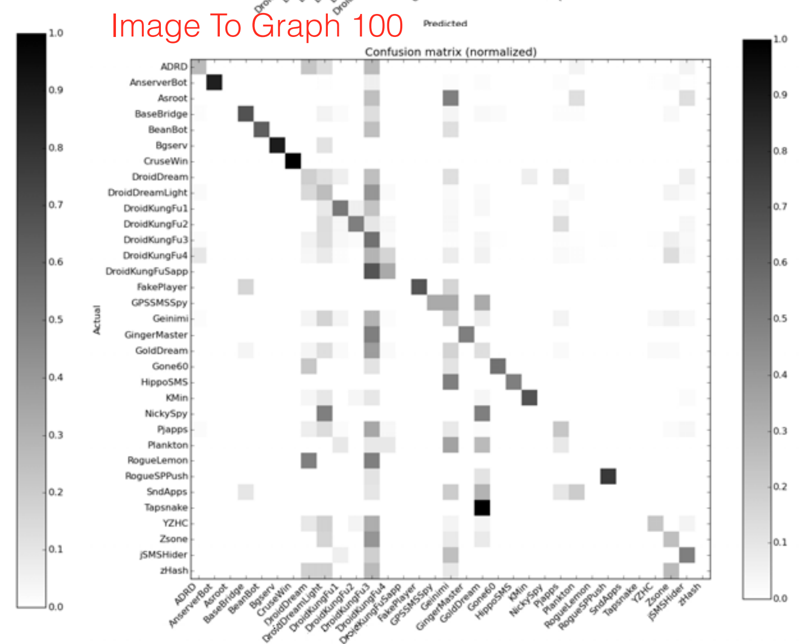


Image To Graph 50

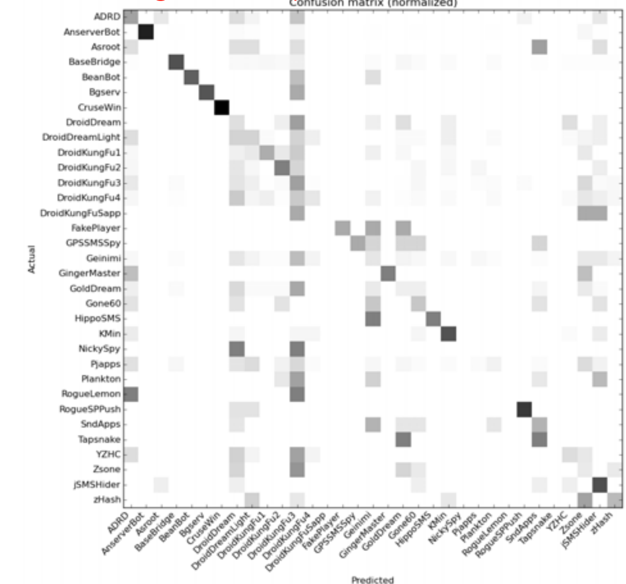


FIGURE 59. IMAGE TO GRAPH + GAUSSIAN NAIVE BAYES

Image To Graph 960

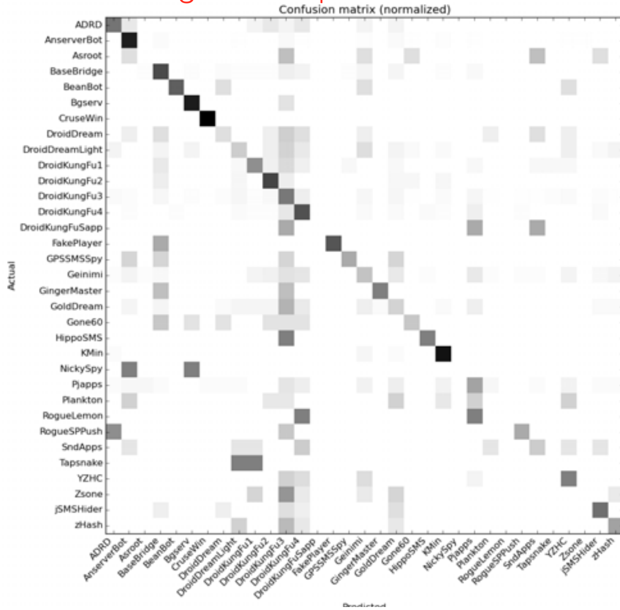


Image To Graph 400

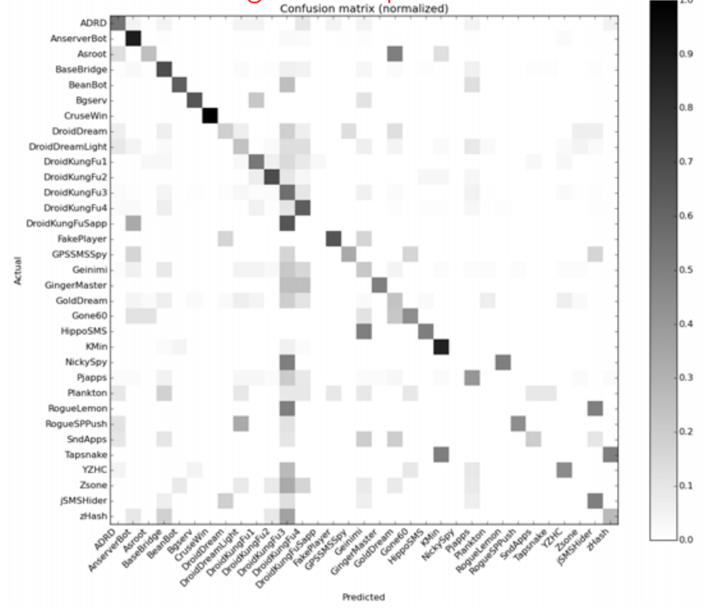


Image To Graph 200

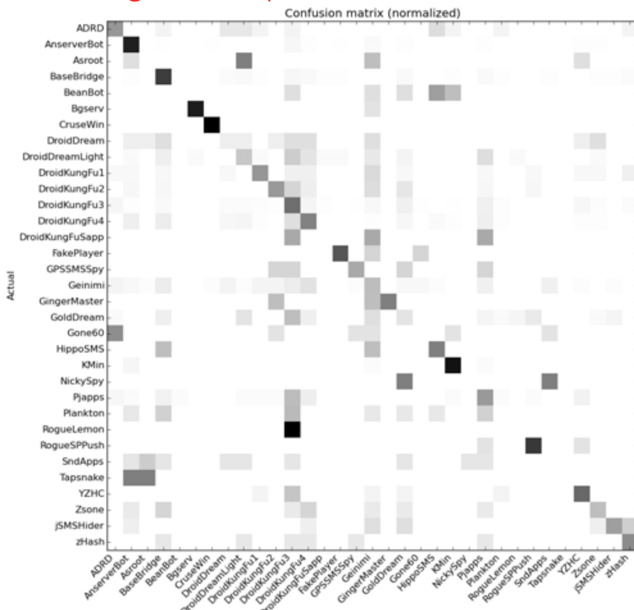


Image To Graph 100

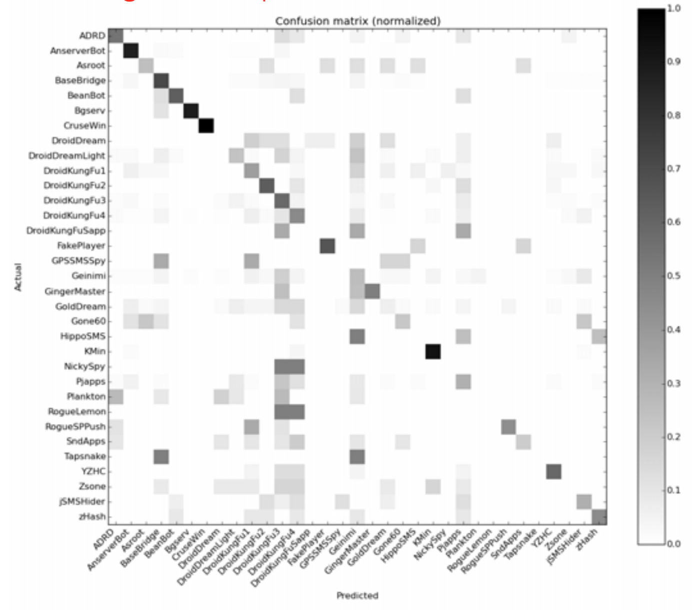


Image To Graph 50

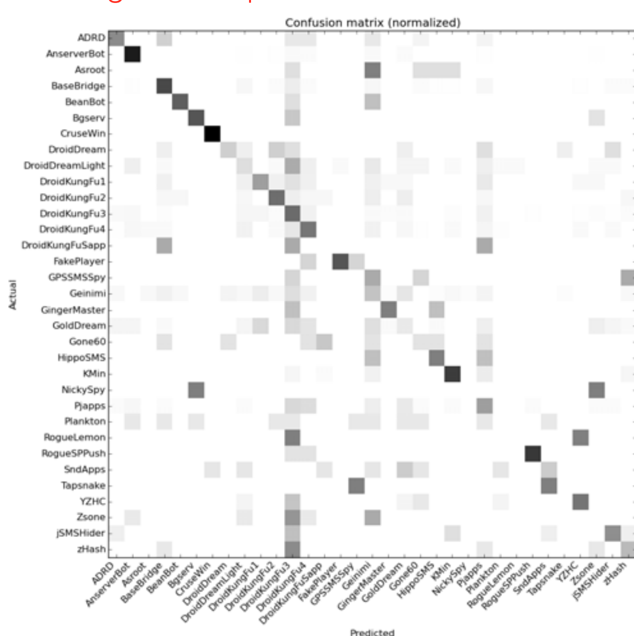
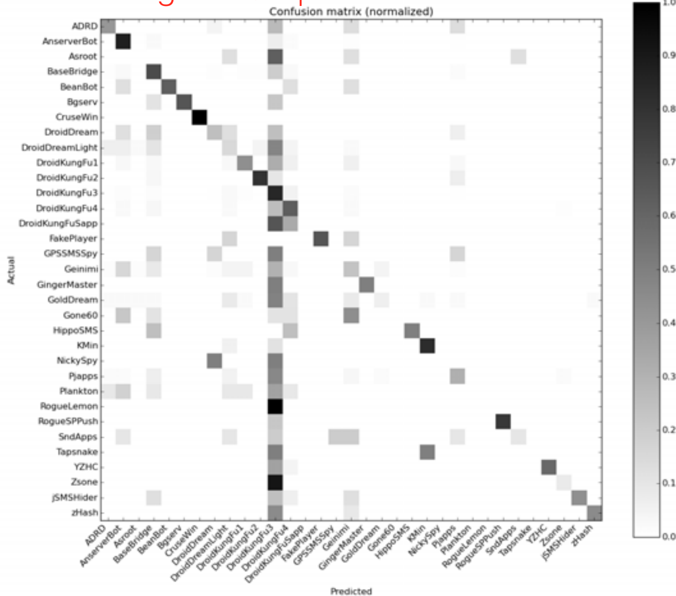
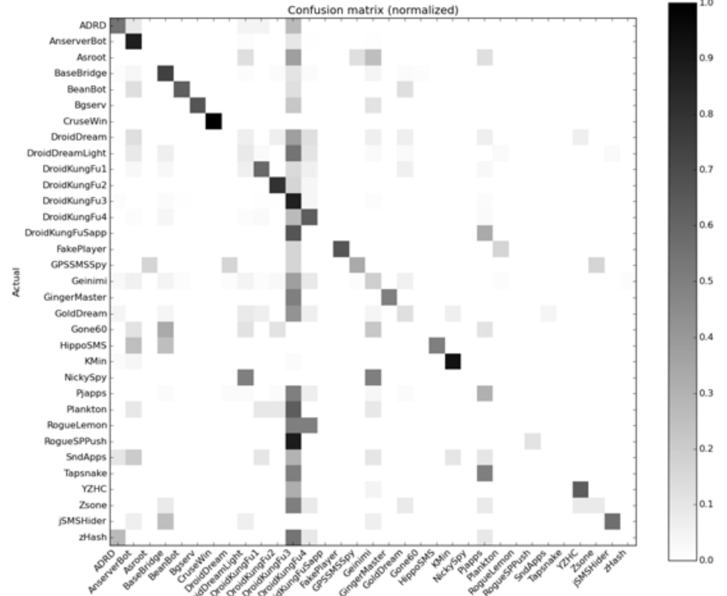
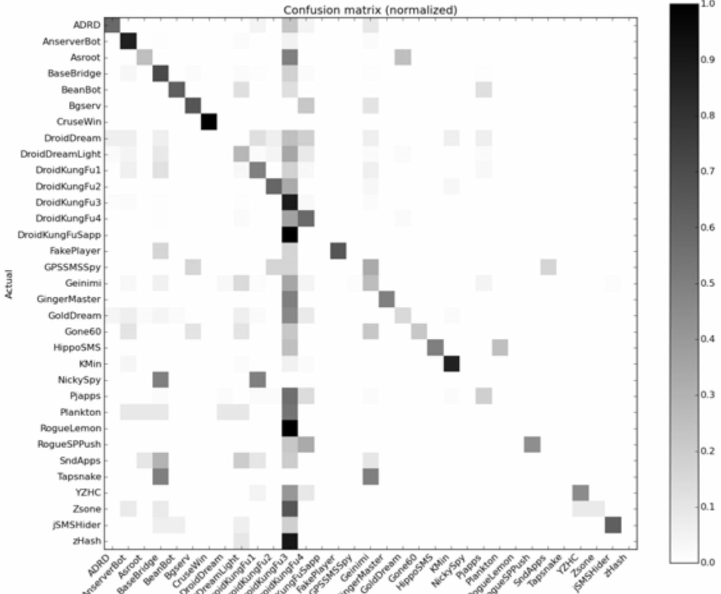
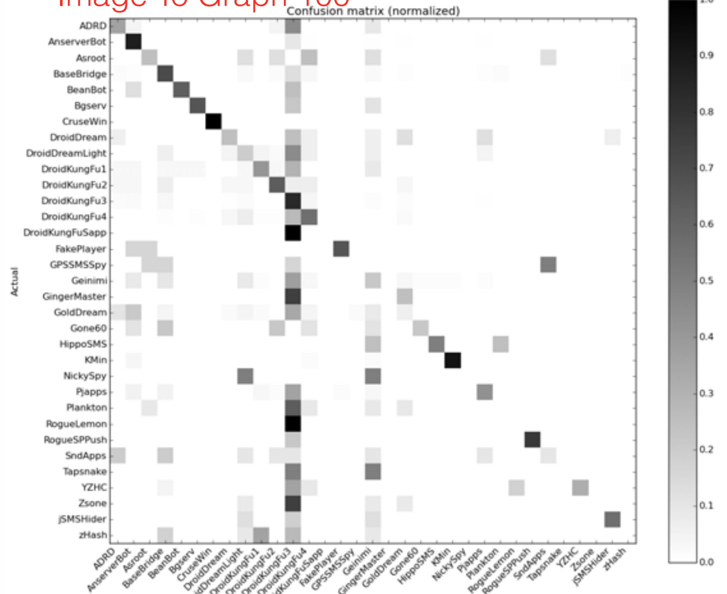


FIGURE 60. IMAGE TO GRAPH + DECISION TREE









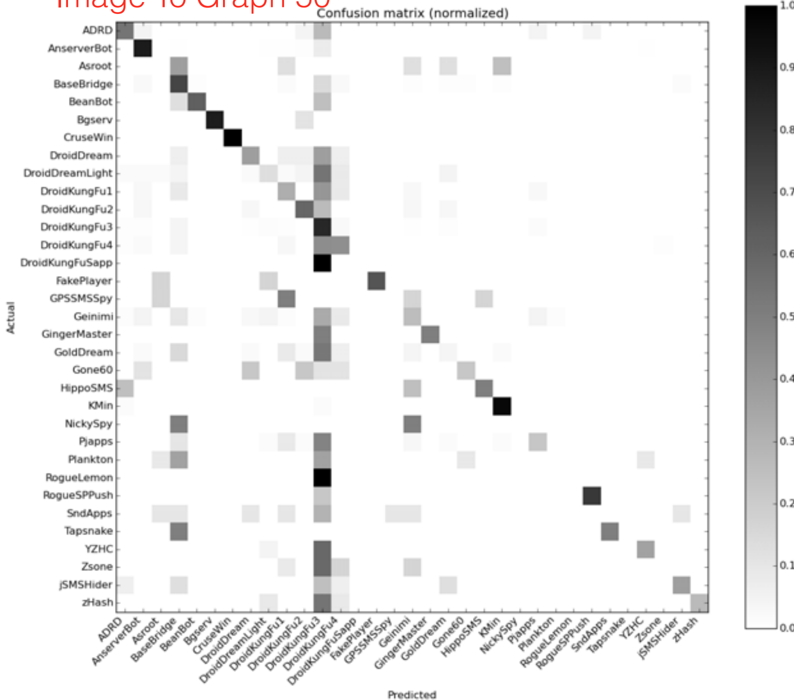
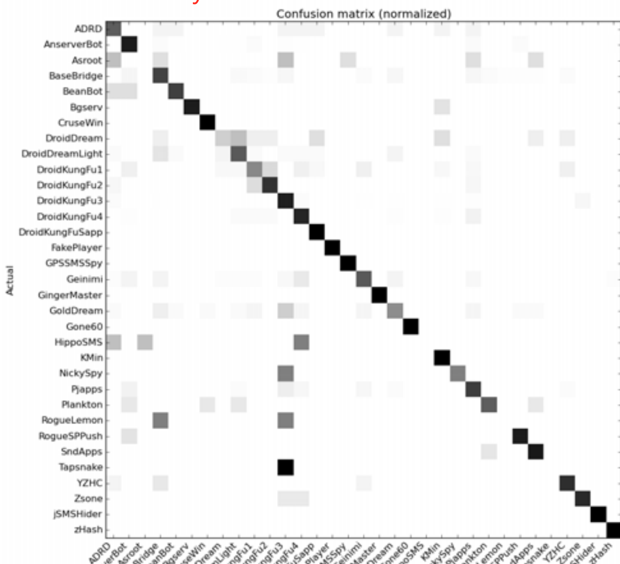
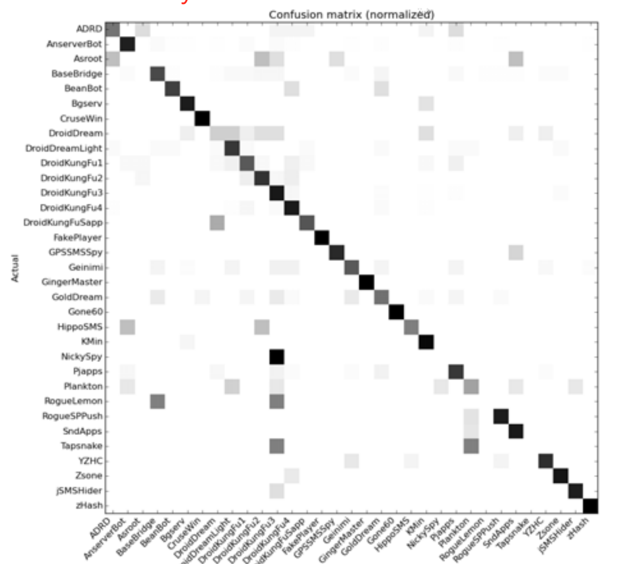
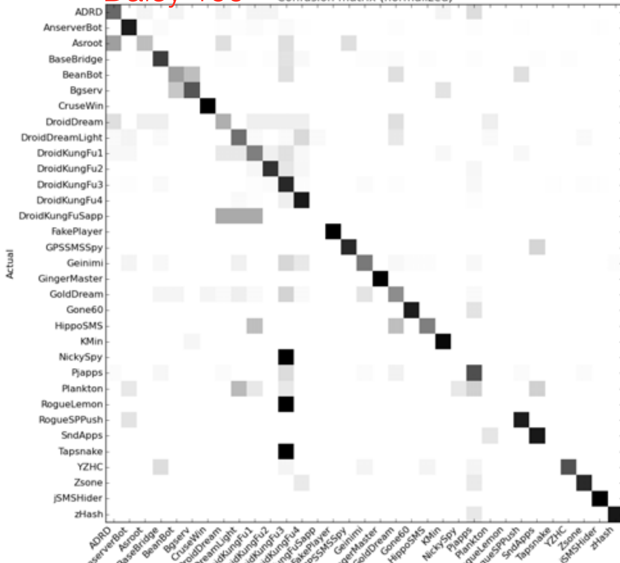
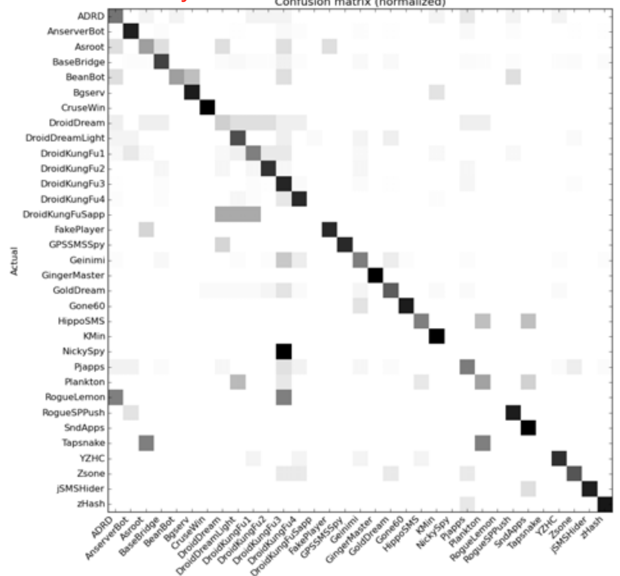


FIGURE 61. IMAGE TO GRAPH + RANDOM FOREST









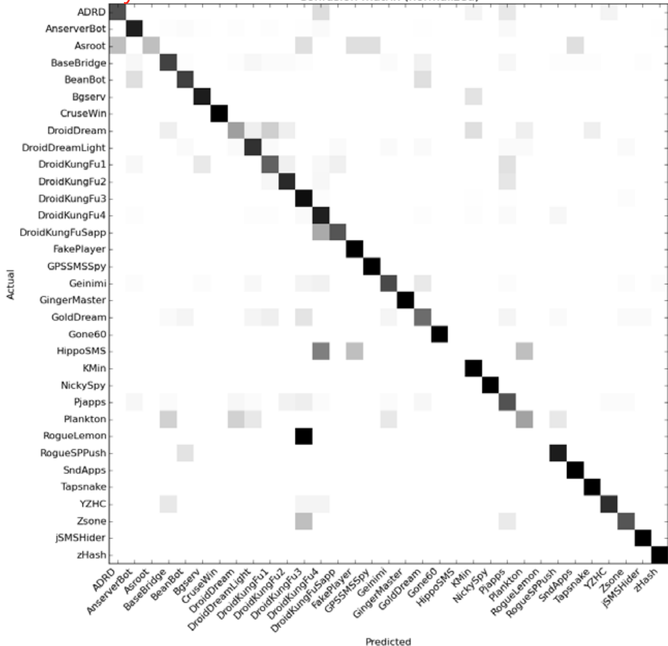
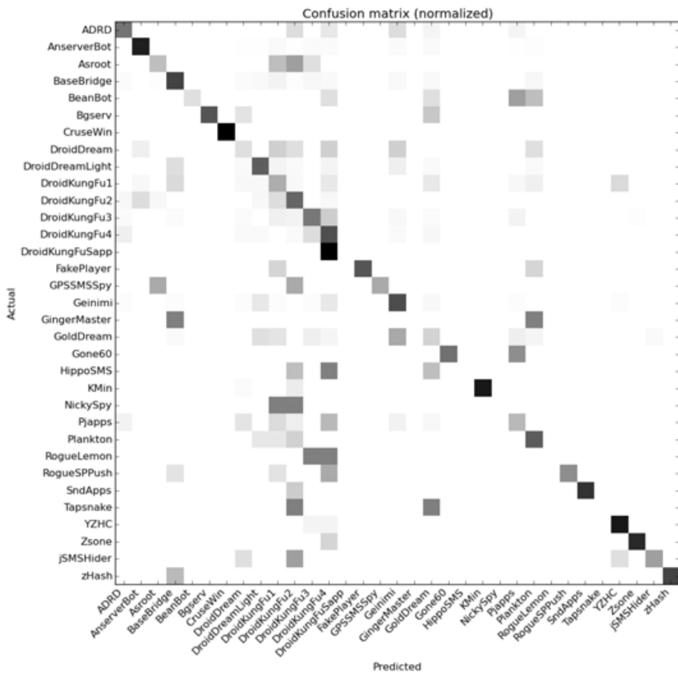
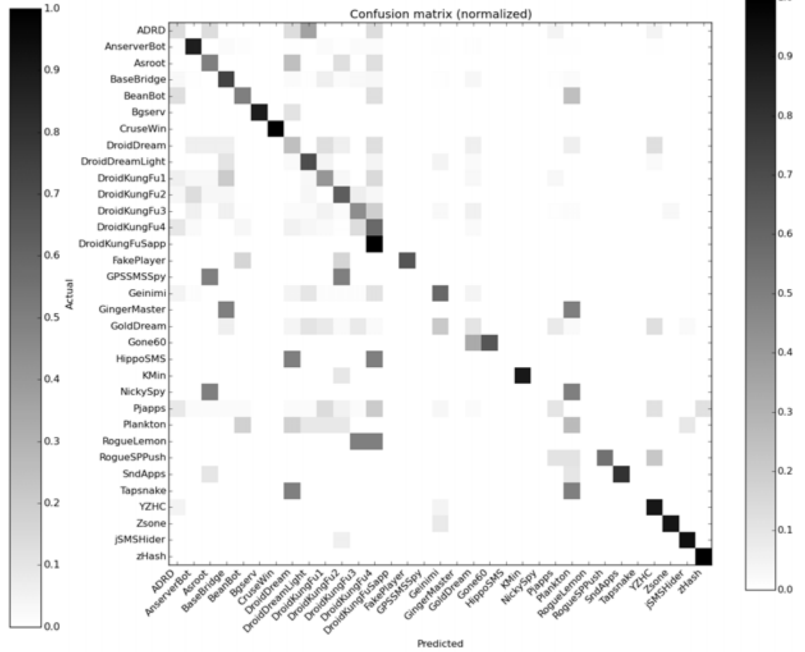


FIGURE 62. DAISY + KNN

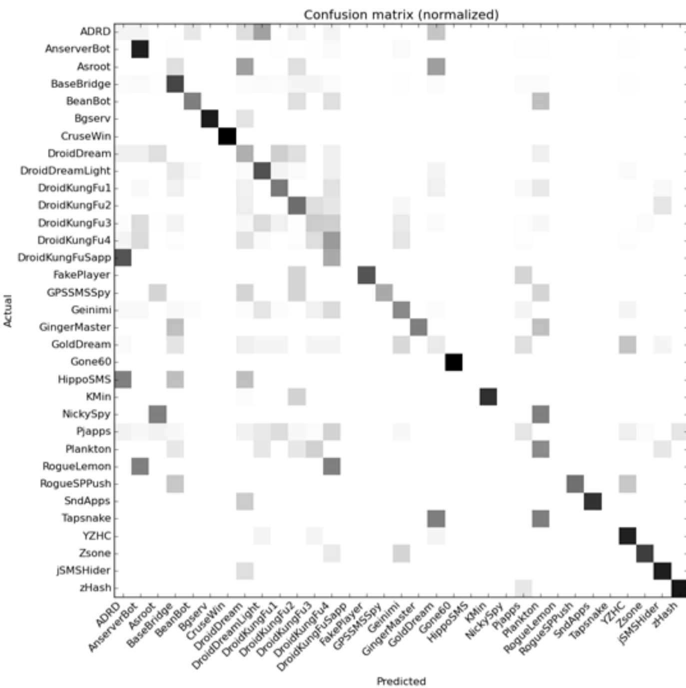
Daisy 400



Daisy 200



Daisy 100



Daisy 50

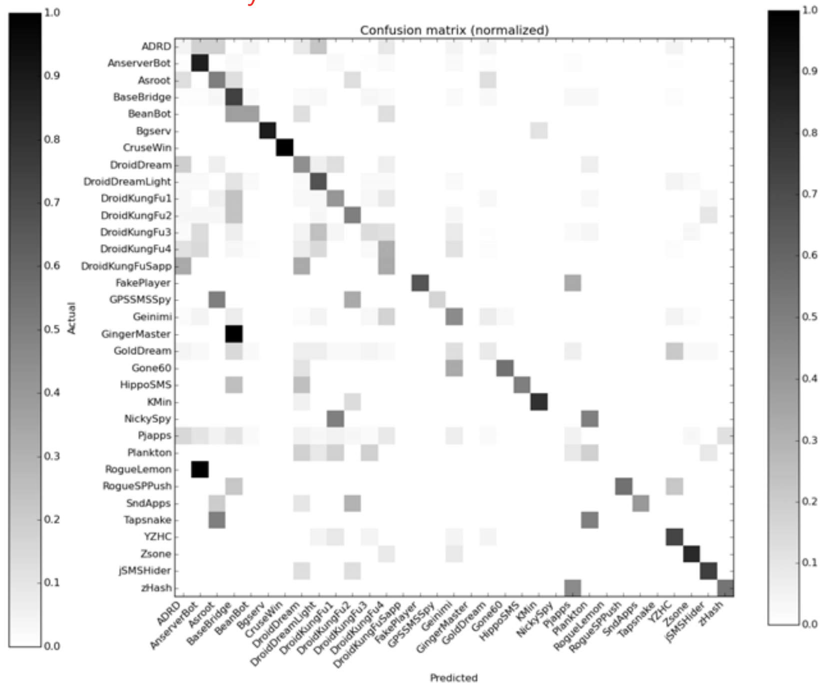
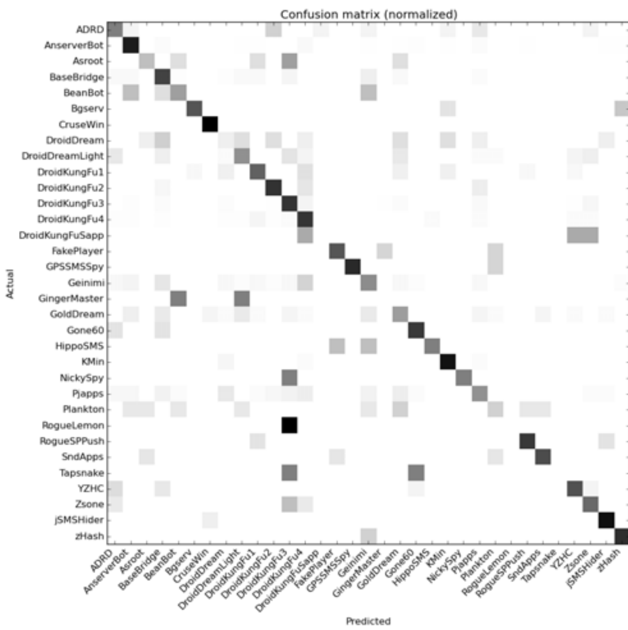
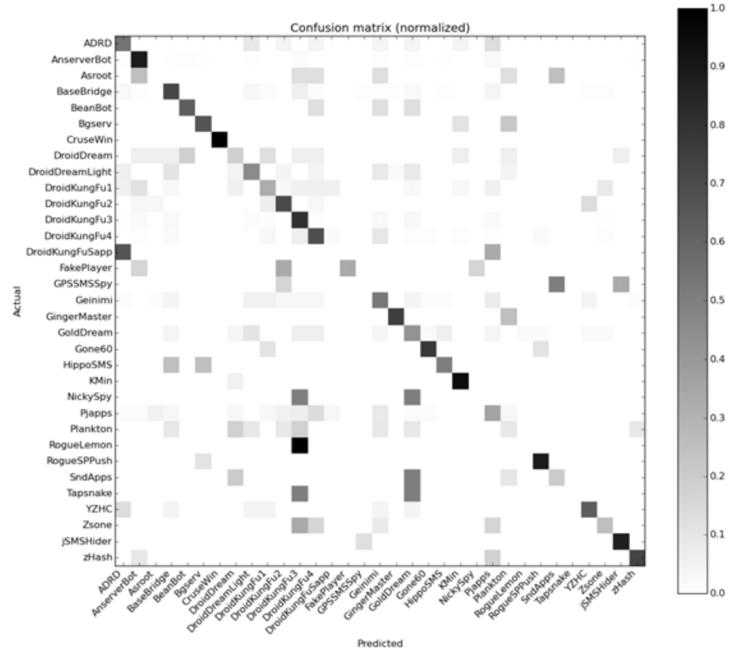


FIGURE 63. DAISY + GAUSSIAN NAIVE BAYES

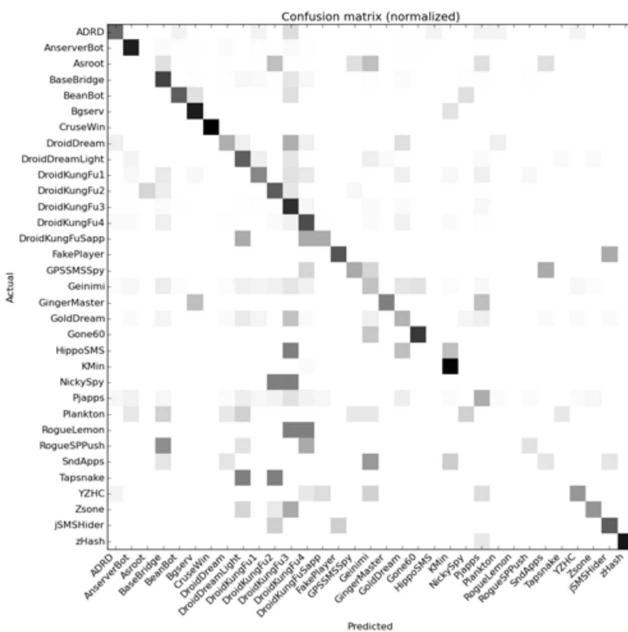
Daisy 400



Daisy 200



Daisy 100



Daisy 50

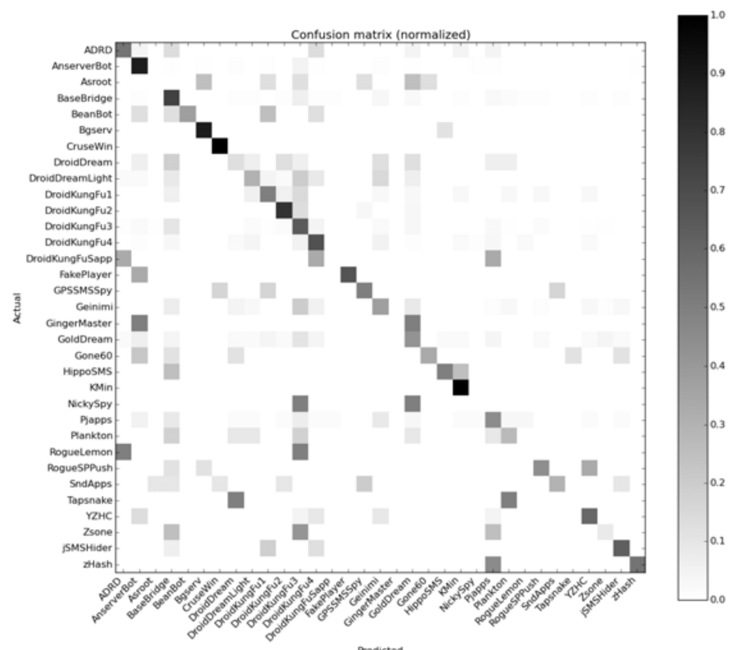
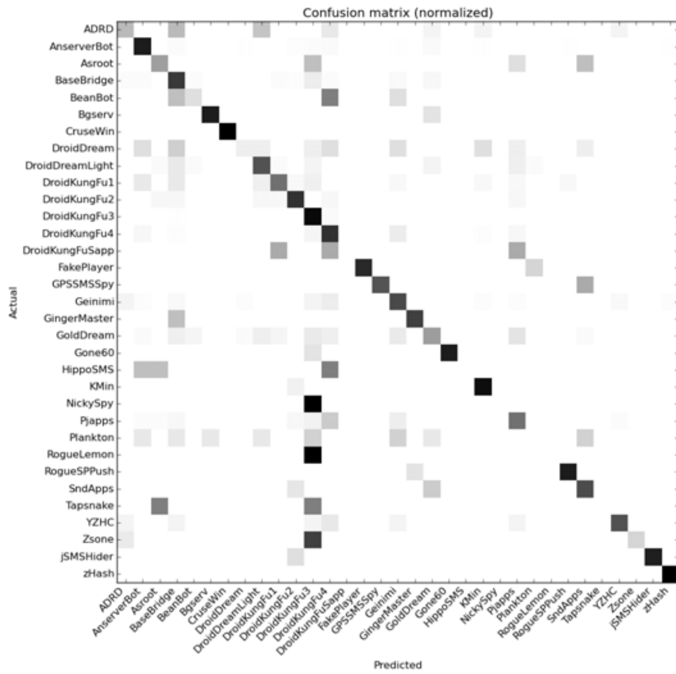
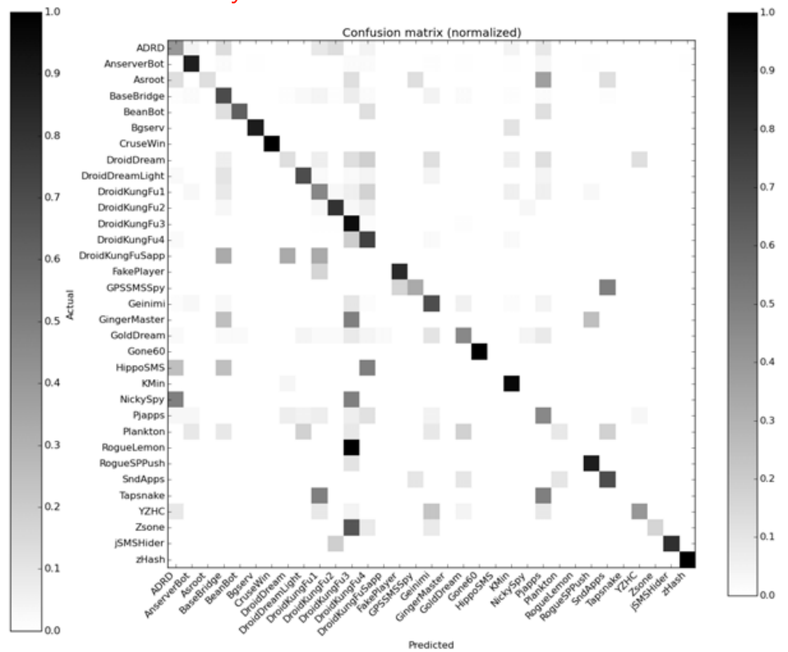


FIGURE 64. DAISY + DECISION TREE

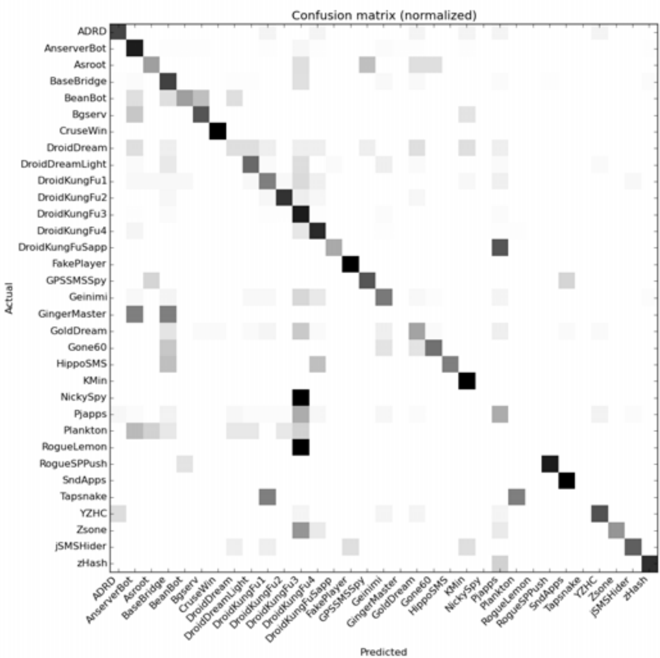
Daisy 400



Daisy 200



Daisy 100



Daisy 50

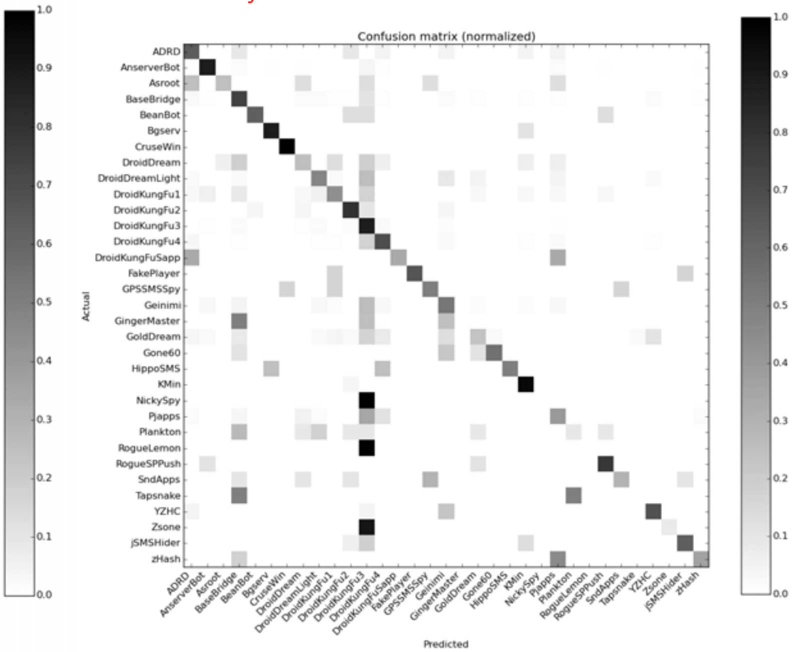


FIGURE 65. DAISY + RANDOM FOREST

Annex III - Time Measurement Table

Time	SIFT(800)	SIFT(400)	GIST(200)	GIST(100)	SIFT(50)	SIFT(400)+PCA	HISTOGRAM(800)	HISTOGRAM(400)	HISTOGRAM(200)	HISTOGRAM(100)	HISTOGRAM(50)	MTOGRAPH(800)	MTOGRAPH(400)	MTOGRAPH(200)	MTOGRAPH(100)	MTOGRAPH(50)	DAISY(800)	DAISY(200)	Daisy(100)	Daisy(50)	Daisy(PRC)+PCA
KNN-1	0.0227341651917	0.00915908613477	0.00292491912642	0.00270795822144	0.00121032796326	0.00117301940918	0.020485162735	0.0093309878303	0.00320867565613	0.00375699996948	0.0012059211731	0.0163290509641	0.0036418586731	0.00287079811096	0.00263810157776	0.00132608413696	0.00435504553931	0.0027631698496	0.0018458968394	0.00119814501135	0.00164004821777
KNN-3	0.02623295784	0.00578498840332	0.00275707244873	0.00163388252258	0.00113916397095	0.000936985015869	0.0220839977264	0.00701308250427	0.00289009643555	0.00166797637939	0.00122904777527	0.0199930087677	0.00572991371155	0.00270050508728	0.00161600112915	0.00114297866821	0.00474381446838	0.00359613990784	0.00153207778931	0.00103868624051	0.000985009924316
SUM - Training	0.0489671230317	0.01494407653809	0.00568199157715	0.00434184074402	0.00235009193421	0.00117301940918	0.0425691604614	0.0163440704345	0.00608897209168	0.00542497634887	0.00243496894837	0.0363221168518	0.01377177238465	0.00557684838378	0.00425410270691	0.00246808280517	0.00909886106789	0.00529932975769	0.00337791442871	0.00222332841186	0.00242614746093
AVG	0.0244835615185	0.00747203826904	0.00284099578857	0.00217092037201	0.00117504596710	0.00117301940918	0.0212845802307	0.00817203521729	0.00304448604584	0.00271248817443	0.00121748447418	0.0181610564259	0.00688588619232	0.00278842449188	0.00212705135345	0.00123405310258	0.00454943053894	0.00264966487884	0.00168895721435	0.00111165142059	0.00121307373046
KNN-1	0.415156841278	0.152417886178	0.0795440673828	0.0425519943237	0.0283029079437	0.00870203971863	0.461503982544	0.18147611618	0.0997610092163	0.0473630428314	0.0261578559875	0.441118001938	0.18176817894	0.0953619480133	0.0542020797729	0.0340578556061	0.0988111495972	0.0501718521118	0.0239239739014	0.0120329856873	0.00958514213562
KNN-3	0.439160108566	0.147042989731	0.0816850390625	0.0427849292755	0.0204031467438	0.00859689712524	0.344142913818	0.182001113892	0.101751089096	0.0487310886383	0.0191090106964	0.429160118103	0.182191133499	0.0991821289062	0.0542228221893	0.031238079071	0.12067484855	0.0506866362427	0.020800113678	0.0120990278337	0.00815105438232
SUM - Testing	0.854316949844	0.299460887909	0.1612091064453	0.0853369235892	0.0487060546875	0.0172999364387	0.805646896362	0.363477230072	0.2015120383123	0.0960941314697	0.0452668666839	0.870278120041	0.363959312439	0.194544076185	0.1084249019622	0.0652959346771	0.2194859381542	0.1008586883545	0.0447239875794	0.024132013321	0.01773619651794
AVG	0.427158474922	0.149730443954	0.0806045532226	0.0426684617996	0.0243530273437	0.00864968421935	0.402823448181	0.181738615036	0.100756041915	0.0480470657349	0.0226334333419	0.435139060020	0.181970656219	0.0972720384597	0.0542124509811	0.0326479673385	0.1097429690771	0.0504293441772	0.0223619937897	0.0120660066605	0.00886809825897
Computing features	15.3796089027	15.2649610043	15.3469980789	15.2109750011	15.226115942	15.2597279549	9.42930912971	9.33618807793	9.322680923	9.33669688276	9.34631490707	10.7949090008	10.663629055	10.6741609573	10.658634901	10.6874678135	16.6005029678	16.5608980656	16.6249001026	16.6771800246	15.747989893
Computing features normalized	0.0123631904362	0.0122708869745	0.0123367347901	0.0122272307082	0.0122396430401	0.0122666623431	0.00757983049012	0.00750497433917	0.00749410042041	0.00750553607940	0.00751311487706	0.00867749876745	0.00857204907958	0.00858051523898	0.00856803448603	0.00859121206872	0.0133445578189	0.0133126190237	0.0133840676069	0.0134060771801	0.0126591558625
Total	18.761	18.081	17.814	17.469	17.455	17.95	12.579	11.905	11.707	11.629	11.565	14.014	13.55	13.045	13.026	12.937	19.076	18.796	18.78	18.822	18.396
Total normalized	0.0150811697106	0.0145345591639	0.0143199356913	0.0140588616720	0.0140313504823	0.0144292604501	0.0101117363344	0.00956993569131	0.00941077170418	0.00934807073954	0.00929662379421	0.0112652733118	0.0108923511254	0.0104860334405	0.0104712668810	0.0103995177684	0.0153344051446	0.0151093247588	0.0150964630225	0.0151302250803	0.0147877813504
NB-1	0.0203850269318	0.0111060142517	0.00677800178528	0.00567507743836	0.0047299861908		0.0206167697906	0.0134208202362	0.00670504570007	0.00516200065613	0.00462198257446	0.0216197957529	0.00960993766765	0.00658582588359	0.00524282455444	0.00456668072205	0.0100650787354	0.00673794746389	0.00548366573792	0.00450992584229	
NB-2	0.0171560314936	0.00967001914978	0.00647497177124	0.00504398345947	0.00458407402039		0.0171409653259	0.00901007652283	0.00651001930237	0.00505708648132	0.00456660788269	0.0169038772583	0.0105290412903	0.00652813911438	0.00508997108459	0.00445103645325	0.00904512405396	0.00656199455261	0.00507307052612	0.00444483757019	
SUM - Training	0.0375430583954	0.0207760334014	0.01325297355652	0.0107190609782	0.00931406021119		0.0377578351165	0.0224306987503	0.0132150650024	0.0102160713745	0.00918869045715	0.0385236740112	0.0201289789581	0.0131396579797	0.0103312795563	0.00899791717153	0.0191102027893	0.0132999420166	0.0105569362640	0.00895476341248	
AVG	0.0187715291977	0.0103880167007	0.00662648677826	0.00535953044891	0.00465703010559		0.0188788175583	0.0112154469751	0.00660753250122	0.00510805486872	0.00459444522857	0.0192618370056	0.0100648947907	0.00656982398988	0.00516639781951	0.00449895858765	0.00955510139468	0.00664997100830	0.00527846813202	0.00447738170624	
NB-1	0.172670841217	0.0662319660187	0.0319221019745	0.0170741081238	0.0103039741516		0.145874023438	0.0524871349335	0.0291919708252	0.0168018341064	0.0100340843201	0.17015004158	0.0642020702362	0.0312261581421	0.017049074173	0.0103170871735	0.0607290267944	0.0301468372345	0.0158768433075	0.0104780197144	
NB-2	0.163384199142	0.0647320747375	0.0298526871265	0.0166320800781	0.0103906693933		0.137735843658	0.0526711940765	0.0272419452667	0.0176289081575	0.00990915298462	0.188095972061	0.0648523377014	0.0302429199219	0.0172989368439	0.0101940631866	0.0597949028015	0.0296928882589	0.0168330669403	0.0101728439331	
SUM - Testing	0.336055040359	0.130968040756	0.061777469101	0.0337061882019	0.0206949710846		0.283609867096	0.10515832901	0.0564336160919	0.0344307422637	0.0199432373047	0.336246013641	0.1290543079376	0.061469078064	0.0343480110169	0.0205111503601	0.120523929596	0.0598397254944	0.0327119102478	0.0206503636475	
AVG	0.168027520179	0.0654820203781	0.0308874844551	0.0168530941001	0.0103474855423		0.141804933548	0.052579164505	0.0282168080459	0.0172153711318	0.00997161865236	0.168123006820	0.0645271539688	0.030734539032	0.0171740055084	0.0102555751800	0.0602619647979	0.0299198627472	0.0163559551239	0.0103254318237	
Computing features	17.5047650337	16.9172258377	16.3494229317	16.3351603707	16.172590971		10.5266320705	9.96166601453	9.60292005539	9.34339785576	9.34440517426	11.7184460163	11.071367979	10.82833004	10.7809400463	10.7654428482	16.6104571819	16.6636829376	16.6715970039	16.6501709031	
Computing features normalized	0.0140713545288	0.0135990561398	0.0131426228354	0.0131311563365	0.0130004750570		0.00846192288625	0.00800777171586	0.00771936811204	0.00751076988051	0.00751157972207	0.00941997288191	0.00889998134679	0.00870444539877	0.00866743891047	0.00865383296479	0.0133524575417	0.0133952435189	0.0134016053086	0.0133843818745	
Total	20.554	19.66	18.883	18.771	18.695		13.532	12.543	11.912	11.524	11.573	14.393	13.64	13.182	12.976	12.991	18.923	18.974	18.831	19.15	
Total normalized	0.0165225080385	0.0158038585209	0.0151792604501	0.0150892282958	0.0150040192926		0.0108778135048	0.0100827974276	0.00957556270096	0.00926366559485	0.00930305466238	0.0115699356913	0.0109646302251	0.0105964630225	0.0104308681672	0.0104429260450	0.0152114147909	0.0152524115756	0.0151374598070	0.0153938906752	
DecisionTree-1	0.552821159363	0.215970089358	0.130569934845	0.052521944046	0.0271918773651		0.0353026774261	0.0274800962666	0.027526140213	0.0221450328827	0.0102889537811	0.608077049255	0.240603023796	0.1079840860	0.0538558959961	0.0301752090454	0.223191022873	0.119429111481	0.0575230121613	0.0269310474396	
DecisionTree-2	0.588726997375	0.259930076324	0.134805099869	0.063784122467	0.0282041091919		0.0359099911285	0.0300509929657	0.0309200288865	0.0233099480602	0.0108561515808	0.593055009842	0.258630037306	0.125308129456	0.0583559532166	0.0274579524994	0.233926057816	0.112137079239	0.0580399336407	0.0293099880219	
SUM - Training	1.14154815673	0.475963115692	0.265475034714	0.116306066513	0.053935986557		0.0712127685546	0.0575110912323	0.0584461688995	0.0454549789429	0.0211451053619	1.20113205967	0.499233961106	0.233290195466	0.1102116492127	0.0576331615448	0.457117080689	0.23156619072	0.115562915802	0.0562410354615	
AVG	0.570774078369	0.237981557846	0.132737517357	0.0581530332565	0.0269679932785		0.0356063842773	0.0287555456161	0.0292230844497	0.0227274894714	0.0105725526809	0.600566025485	0.249615980553	0.116645097733	0.0551059246063	0.0288165807724	0.228558540344	0.11578309536	0.0577814579001	0.0281205177307	
DecisionTree-1	0.010929107666	0.00116610527039	0.00076985359192	0.00046014785767	0.00033712387085		0.0018507491548	0.00094413757324	0.00070810317993	0.00043511390686	0.00032496452332	0.0023181438446	0.00262498855591	0.00082111358643	0.00061011314392	0.00037813186646	0.00083112716675	0.00059795379639	0.00043106079102	0.00033593177795	
DecisionTree-2	0.00164580345154	0.000859348654175	0.00060606002808	0.00040102005006	0.00031304359436		0.00158500671387	0.00092315673828	0.00072312355042	0.00037884712219	0.00032496452332	0.00155319442749	0.00087499618530	0.00061011314392	0.00043201446533	0.00032496452332	0.00079798698425	0.000593900680864	0.00038290023804	0.00034189224243	
SUM - Testing	0.0125749111754	0.00116610527039	0.0013751362	0.00086116790772	0.00065016746521		0.0032200918329	0.0018672934112	0.00143122673035	0.000813961036805	0.000649929046	0.00387133827209	0.00349998474121	0.00143122673035	0.00104216766925	0.00070309638978	0.00162911415	0.00119185447695	0.00081396102905	0.00067782402038	
AVG	0.00628745555877	0.00116610527039	0.00068795681	0.00043058395386	0.000325083732605		0.00161004066475	0.00093364715576	0.000715613365175	0.000406980514525	0.00032496452332	0.00193566913604	0.00174999237060	0.000715613365175	0.00052106380462	0.00035154819489	0.00081455707550	0.000595927238465	0.00040696051453	0.00033891201019	
Computing features	15.6677081585	15.7366440296	16.6533329487	15.2398769855	15.2352540493		12.2560300827	9.80591201782	9.60617399216	9.3708900452	9.34678220749	10.8212940693	10.8013961315	10.8601942062	10.7091310024	10.8184578419	16.7587170601	16.644370079	16.6160340309	16.5897648335	
Computing features normalized	0.0125946207061	0.0126500353935	0.0133866235920	0.0122507049722	0.0122469887856		0.00985211421438	0.00788256593072	0.00772204816848	0.00755303006794	0.00751349052049	0.00869878944477	0.00868274317926	0.00873005985128	0.00860862627717	0.00869650591921	0.0134716375061	0.0133797187130	0.0133569405393	0.0133358238211	
Total	19.071	18.602	19.252	17.52	17.478		14.728	13.264	11.873	11.596	11.525	14.316	13.569	13.524	12.982	13.049	19.463	19.088	18.938	18.747	
Total normalized	0.0153303858521	0.0149533762057	0.0154758842443	0.0140836012861	0.0140498392282		0.0118392282958	0.0106625080321	0.00954421221865	0.00932315112540	0.00926446945337	0.0115080385852	0.0109076366559	0.0108713826366	0.0104356913183	0.0104895498392	0.0156157556270	0.0153440514469	0.0152234726688	0.0150699356913	
RandomTree-1	0.129956007004	0.091010093689	0.071051161493	0.0534651273449	0.0426599848022		0.0352720489502	0.0328629016876	0.0352160930634	0.0383380455017	0.030445098877	0.139333963394	0.093418029245	0.0705095854208	0.0528320240021	0.0440630065338	0.0894529819489	0.0685625844666	0.0568660431366	0.0435919761658	
RandomTree-2	0.120804071426	0.0929027825928	0.0672486213585	0.0533289609863	0.0419559015503		0.0329811301422	0.0338590014511	0.0340650081635	0.0360569953918	0.031144142150	0.129590034485	0.0926369634214	0.0701110363007	0.0556619167328	0.0442659854889	0.0877900123596	0.0657250881195	0.0534710950094	0.0451841490173	
SUM - Training	0.25076007843	0.1840028762818	0.1383299827575	0.1067941188812	0.0846178863525		0.0692331790924	0.0667219161987	0.0692811012268	0.0749950408935	0.0615692410279	0.268923997679	0.1860558986664	0.1407198905945	0.1085939407349	0.0883297920227	0.1772429943085	0.1342860725861	0.110139131546	0.0887861251831	
AVG	0.12538003921	0.092001438140	0.0691649913787	0.0533970594406	0.0423089317625		0.0346165895462	0.0333608580993	0.0346405505134	0.0374975204467	0.0307846205139	0.134461998839	0.0930279493332	0.0703599452972	0.0542969703674	0.0441648960113	0.0886214971542	0.0671440362930	0.055069565773	0.0443930625915	
RandomTree-1	0.00587391853333	0.00637984275818	0.00466418266296	0.00419116020203	0.00398802757263		0.0060570240320	0.00507402420044	0.0048560784312	0.0045900344848	0.00532603283855	0.00616097450256	0.00498009728027	0.00466585159302	0.00434517980413	0.00407290456579	0.00485603054810	0.00436806678772	0.0042130947113	0.00393009185791	
RandomTree-2	0.00537014007568	0.00457215309143	0.00399486022949	0.00372895622852	0.00357103347778		0.00533795535675	0.00458407402039	0.00402116775513	0.0036730796296	0.00370001792908	0.00564592903717	0.00450601849365	0.00445699691772	0.00374364741211	0.00364995032747	0.00413417816162	0.00378296043396	0.00369811058044	0.00378108024597	
SUM - Testing	0.01124405860901	0.01095139584951	0.00865904289245	0.00791811943055	0.00755906105041		0.0113949775696	0.00965809822083	0.0086772441864	0.00826311114	0.00902605056763	0.0118069645742	0.00948810577392	0.00912284851074	0.00808502601624	0.00772285451426	0.00899022087872	0.00815102722168	0.00791120529174	0.00771171210388	
AVG	0.00562202990450	0.00547569792480	0.00432952144622	0.00395905971527	0.00377953052520		0.0056974887848	0.00482904911041	0.00433862209320	0.0041315555725	0.00451302528381	0.00590348243713	0.00474405288696	0.00456142425537	0.00404251300812	0.00386142730713	0.00449511043545	0.00407551361084	0.00395560264587	0.00385585605194	
Computing features	16.1518158913	15.9153590202	15.4831919861	15.4978969097	15.3126428127		9.69739603995	9.42628002167	9.40506100655	9.48553800583	9.58952307701	10.9070448875	10.8099319995	10.682860136	10.7220010757	10.73843503	16.6301500797	16.5991458893	16.6509571705	16.7891039848	
Computing features normalized	0.0129837748322	0.0127939696915	0.0124302186383	0.0124581164868	0.0123091984024		0.00779533443726	0.00757735515811	0.00756033842970	0.00762503055131	0.00770861989868	0.00876772097065	0.00868965594332	0.00858750814700	0.00861897182580	0.0086321825	0.0133682678454	0.0133433646862	0.0133922465293	0.0134960684229	
Total	18.736	18.549	17.933	18.001	17.689		46.763	11.843	11.675	11.844	45.609	13.603	13.272	13.182	13.1	13.078	18.938	19.008	18.987	19.133	
Total normalized	0.0150610932475	0.0149107717041	0.0144155948553	0.0144702572347	0.0142194533762		0.0375900835012	0.00952009646302	0.00938504823151	0.00952090032154	0.0366627491961	0.0109348874598	0.0106688102894	0.0105964630225	0.0105305466238	0.0105128617363	0.0152234726688	0.0152797427653	0.0152628617363	0.0153800225080	

Table 21. Time Measurement (seconds)