UNIVERSITÀ DEGLI STUDI DI GENOVA

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Android Malware Detection Using Network Behavior Analysis And Machine Learning Classifiers

Supervisor:

Prof. Gabriele Costa

Candidate:

Hamidreza Aria

Student n. S3819346

March 2017

List of Figures

List of Tables

Chapter 1

Preface

1.1 Introduction

For many people, mobile phones are essential tools for the everyday life. According to International Data Corporation[1] (IDC), Android OS is the most popular smartphone platform (86.8% of the market share), followed by apple iOS (12.5%). in the Third quarter of 2016 Statistically [1] speaking, it is also the first targeted platform by malware authors seeking to take the control over millions of Android smartphones over the world. It is therefore necessary to pay special attention to aspects related to the Android platform security.

Android is an open source operating system. The developers rely on a software development kit (SDK) to build and publish their applications. Applications are released through an official app store called Google Play. The popularity of Android is a result of being an open source, third-party distribution centers, a rich SDK, and the popularity of Java as a programing language. Due to this open environment, Android is the mobile platform that has been targeted the most by malware that aim to steal personal information or to control the user's devices; In addition Android has various third party application stores which makes it easy for cybercriminals to repackage Android applications with piece of malicious code. Besides, exploiting vulnerabilities in the platform, hardware, or other installed apps to launch malicious behaviors. Mainly, malware authors seek

[1] http://www.idc.com/promo/smartphone-market-share/os

access confidential data a use of device, monetary benefits via premium SMS, or joining the device to a botnet or etc.

Research studies in the Android malware detection field work in two approaches: static, dynamic Analysis. In static analysis, malware is disassembled into a source code from where specific features are extracted. In dynamic analysis, malware is monitored at run-time in isolated environment. In the both approaches, machine learning algorithms have been used to build classification models by training classifiers with datasets of malware and features that collected from static or dynamic analysis. The learned classification models are then used to detect malicious apps and classify them into their families.

1.2 Motivation

Smart mobile devices have been widely used and lots of sensitive information is saved in smartphones. A huge number of malware is being developed for stealing the private data or encrypting or deleting sensitive data, altering or hijacking core computing functions and monitoring and monitoring user's computer activity without their permission. Nowadays people store many of their confidential data, financial information and personal content like photos, videos and confidential data like bank details, usernames and passwords in their smartphones, so these smartphones become more interesting and an attacker can attack on their devices to steal their information. Due to the popularity of Android OS[2], attack on Android means attacking many devices**.** According to a report by Kaspersky Labs, there were 291,800 new mobile malware programs that emerged in the second quarter of 2015, which is 2.8 times more than in the first quarter. In addition, there were 1 million mobile malware installation packages in the second quarter of 2015, which is 7 times more than the first quarter of 2015 [2]. Due to this alarming increase in the number of Android malware applications, the analysis and detection of Android malware has become an important research area. Some of the malwares are more dangerous than the others

[2] http://www.idc.com/promo/smartphone-market-share/os

because of the fact that they connect to some remote server in the background to get commands or to leak/send private information of the users or device itself to the server. None of the techniques proposed so far have focused on these types of malwares. Botnets have become one of the major threats on the Internet for serving as a vector for carrying attacks against organizations and committing cybercrimes. They are used to generate spam, carry out DDOS attacks and click-fraud, and steal sensitive information.

Hence it is crucial to have tools for detecting Android malware, we decide in this thesis analyze network traffic behavior of Android applications for detect botnets malwares from benign applications, then in next step we detect family these type of Android malwares .

1.3 Scope

In this thesis we carry out a systematic analysis of machine learning-based method applied to the problem of identifying dynamic malware behavior that uses the features that can be extracted from the Network log files of the Android applications. The thesis is divided in two major parts; in the first part we propose binary classification methods for discovery or detection of mobile Android from benign applications; to achieve this goal we propose and apply two approaches in base on machine learning algorithms and finally compare the results obtained with each other. And in the second part we are going to classification of malware in a base on families that belong to it. To this aim we propose two method in a base of supervised machine learning algorithms for classification and detection of malwares families.

In the second part of our thesis we aim to present supervised machine learning approaches to detection of botnets malware on base of their families. Fortunately we select a malware dataset [40] that it gives us allowed to perform research of this topic. For achieve this aim we focusing on HTTP protocol in this way recognize family of malwares in our dataset.

1.4 Thesis structure

The structure of this thesis is organized as follows. Chapter 2 provides the context of the Android operating system and then theoretical background about some relevant network protocols like TCP, UDP and HTTP. Chapter 3 presents related work about the static and dynamic malware detection in Android environment. Chapter 4 shows the implementation of this study which covers the framework, used tools, datasets, extracting and selecting features, and training machine learning classifiers. Chapter 5 demonstrates the results, and performance evaluation of the classifiers. Chapter 6 concludes the study work, emphasizes our findings, and suggests further potentials for future work for our proposed approaches in this thesis.

Chapter 2

Background

In this chapter, we detail background information required to understand our analysis on botnet malwares. The theoretical background about the basic methods and components that make up this thesis is described in follow. Many traffic features have been used for detection of malicious activity in network security. Some of the features used in the proposed work selected based upon TCP, UDP and HTTP flows, knowing these concepts will help us better understand our methodology in Section 4; but first we discuss about Android ecosystem.

2.1 Android Architecture

Android is an open source, Linux-based software stack created for a wide array of mobile devices. As visible in Figure 1 shows the major components of the Android platform[3].

On top of the stack architecture of Android are system applications that offer the basic functionality (management email, calendar etc.). Exactly below it there is a Java API framework, which provides a set of reusable modular components for the development of new applications in the Java language, such as functions to access resources or to show notifications in the status bar of the app. The Android system also makes available access to native libraries written in C / C ++ through the use of Android NDK (Native Development Kit), in case you want to achieve higher

[3] http://developer.android.com/guide/platform/index.html

performance in transactions involving the use of 2D and 3D graphics. Android Runtime is the level where there are multiple instances of virtual machines (one per application) optimized to have a low consumption of resources. In particular, each virtual machine running the DEX bytecode generated, starting from the app written in Java**.** Hardware Abstraction Layer is formed by a set of libraries that makes possible the utilization of hardware resources by the upper level (Java API framework). The basis of all architecture is the Linux Kernel, which abstracts the hardware of the device and provides basic functionality, such as for the management of processes, memory or Inter Process Communication (IPC).

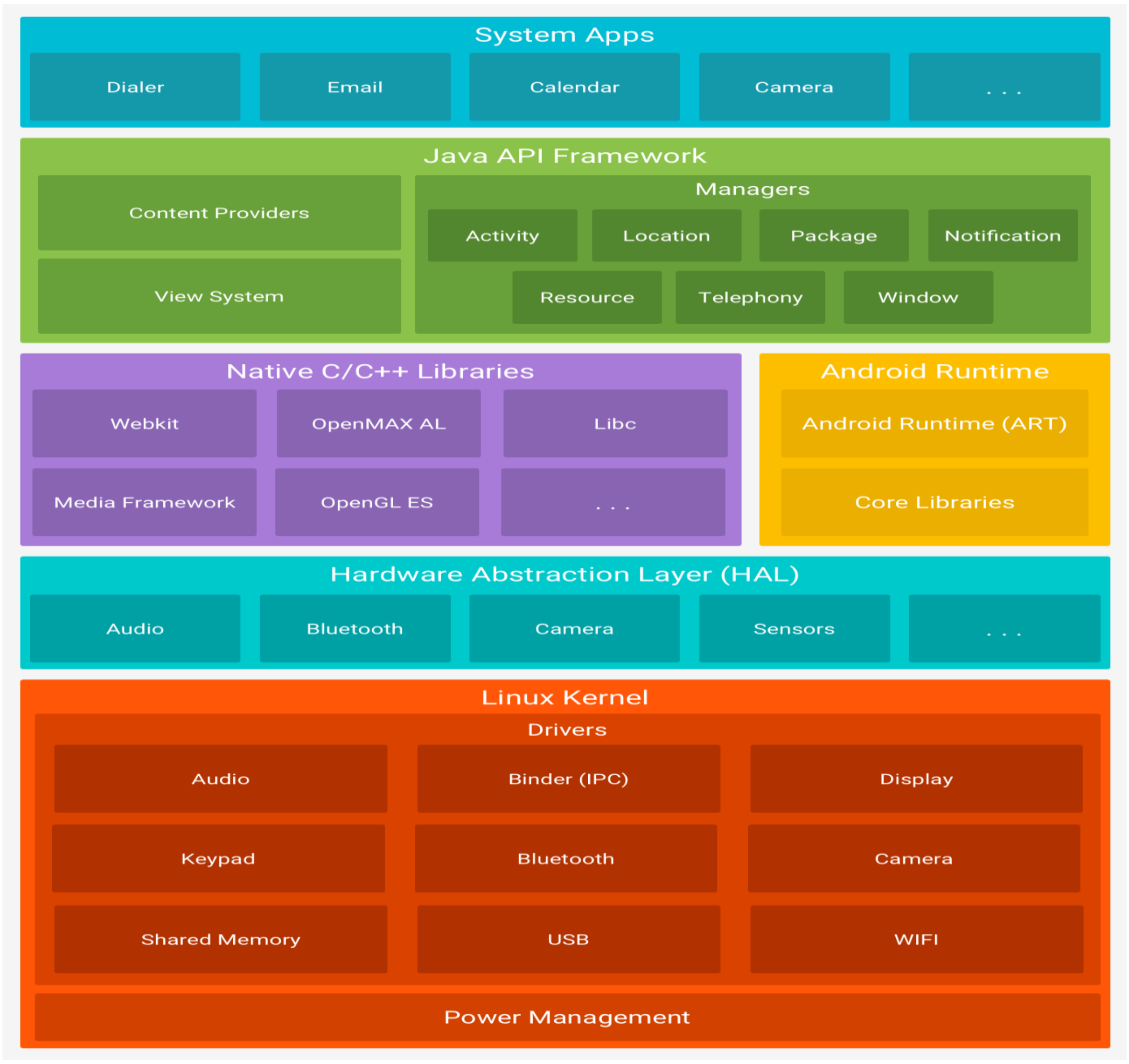


Figure 1: Android architecture platform

The Android operating system assigns to every application a unique Linux user ID during installation phase In combination with the fact that each app runs in its own instance of virtual machine, this contributes to the creation of a sandbox which guarantees isolation of each installed application than the others and respect to the operating system Android implements the principle of least privilege[4], so any app that wants access to protected resources of the device (camera, external memory etc.) or to access data from other applications must explicitly request for user authorization. This authorization mechanism is guaranteed in Android through the use of permissions. The previous version of android 6.0 permissions are required during installation of the app and the user must accept all of them until that the installation is completed[5] in later versions permissions are managed dynamically (that are required when used).

2.2 TCP Flow

Transmission Control Protocol (TCP) is a standard that defines how to establish and maintain a network conversation via which application programs can exchange data. TCP works with the Internet Protocol (IP), which defines how computers or devices send packets of data to each other. Together, TCP and IP are the basic rules defining the Internet. TCP is a connection-oriented protocol, which means a connection is established and maintained until the application programs at each end have finished exchanging messages. It determines how to break application data into packets that networks can deliver, sends packets to and accepts packets from the network layer, manages flow control, and because it is meant to provide error-free data transmission handles retransmission of dropped or garbled packets as well as acknowledgement of all packets that arrive. In the Open Systems Interconnection (OSI) communication model, TCP covers parts of Layer 4, the Transport Layer, and parts of Layer 5, the Session Layer. For example, when a Web server sends a HTML file to a client, it uses the HTTP protocol to do so. The HTTP program layer asks

[4] http://developer.android.com/guide/components/fundamentals.html

[5] http://developer.android.com/training/permissions/requesting.html

the TCP layer to set up the connection and sends the file. The TCP stack divides the file into packets, numbers them and then forwards them individually to the IP layer for delivery. Although each packet in the transmission will have the same source and destination IP addresses, packets may be sent along multiple routes. The TCP program layer in the client computer waits until all of the packets have arrived, then acknowledges those it receives and asks for the retransmission on any it does not (based on missing packet numbers), then assembles them into a file and delivers the file to the receiving application. Retransmissions and the need to reorder packets after they arrive can introduce latency in a TCP stream. Highly time-sensitive applications like voice over IP (VoIP) and streaming video generally rely on a transport like User Datagram Protocol (UDP) that reduces latency and jitter (variation in latency) by not worrying about reordering packets or getting missing data retransmitted.

2.2.1 Connection establishment

To establish a connection, TCP uses a three-way handshake. Before a client attempts to connect with a server, the server must first bind to and listen at a port to open it up for connections this is called a passive open. Once the passive open is established, a client may initiate an active open. To establish a connection, the three-way (or 3-step) handshake occurs (see Figure 2):

1) SYN: The active open is performed by the client sending a SYN to the server. The client sets the segment's sequence number to a random value A.
2) SYN-ACK: In response, the server replies with a SYN-ACK. The acknowledgment number is set to one more than the received sequence number i.e. A+1, and the sequence number that the server chooses for the packet is another random number, B.
3) ACK: Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value i.e. A+1, and the acknowledgement number is set to one more than the received sequence number i.e. B+1.

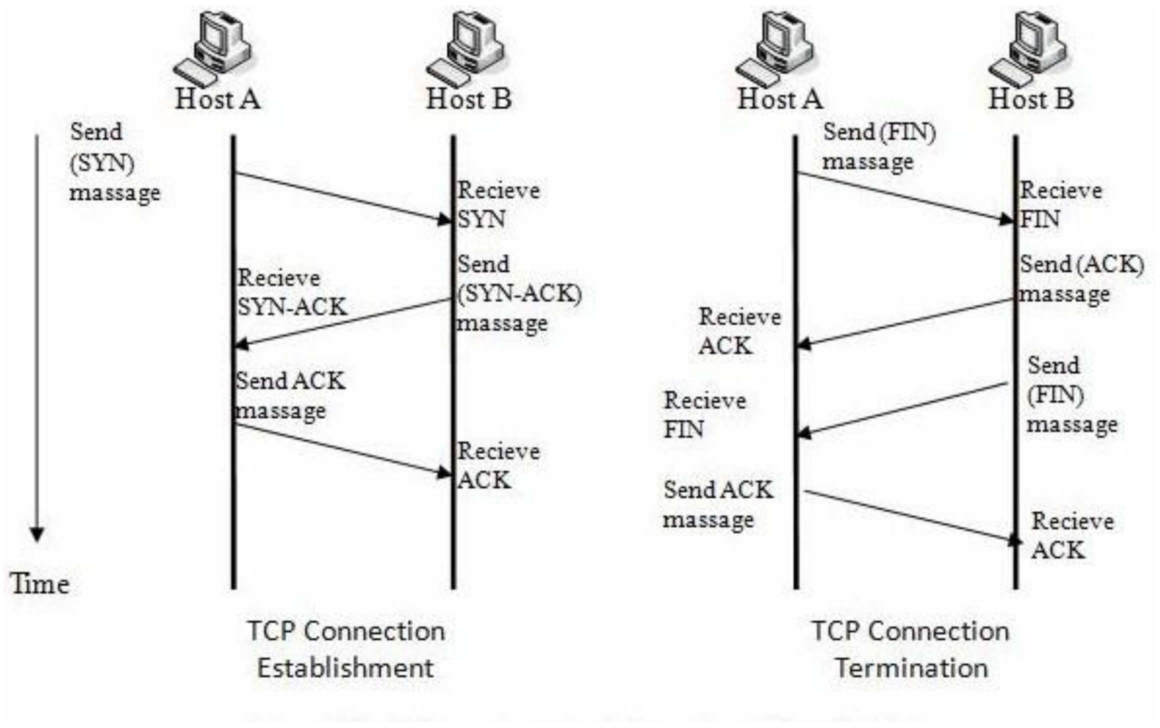


Figure 2: Three-Way-Handshake TCP protocol

At this point, both the client and server have received an acknowledgment of the connection. The steps 1, 2 establish the connection parameter (sequence number) for one direction and it is acknowledged. The steps 2, 3 establish the connection parameter (sequence number) for the other direction and it is acknowledged. With these, a full-duplex communication is established.

2.2.2 Connection termination

The connection termination phase uses a four-way handshake, with each side of the connection terminating independently. When an endpoint wishes to stop its half of the connection, it transmits a FIN packet, which the other end acknowledges with an ACK. Therefore, a typical tear-down requires a pair of FIN and ACK segments from each TCP endpoint. After the side that sent the first FIN has responded with the final ACK, it waits for a timeout before finally closing the connection, during which time the local port is unavailable for new connections; this prevents confusion due to delayed packets being delivered during subsequent connections. It is also possible to terminate the connection by a 3-way handshake, when host A sends a FIN and host B replies with a FIN and ACK (merely combines 2 steps into one) and host a replies with an ACK. For a program flow like above, a TCP/IP stack like that described above does not guarantee that all the data arrives to the other application if unread data has arrived at this end.

2.3 UDP flow

In the Open System Interconnection (OSI) communication model UDP like TCP is in layer 4, The Transport Layer. UDP works in conjunction with higher level protocol to help manage data transmission services including trivial file transfer protocol (TFTP), Real Time Streaming Protocol (RTSP), Simple Network Protocol (SNP) and Domain Name System (DNS) lookups.

UDP uses a simple connectionless transmission model with a minimum of protocol mechanism. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network and so there is no guarantee of delivery, ordering, or duplicate protection. If error correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose. UDP is an ideal protocol for network applications in which perceived latency is critical such as gaming, voice and video communications, which can suffer some data loss without adversely affecting perceived quality. In some case, forward error correction techniques are used to improve audio and video quality in spite of some loss. UDP can also be used in applications that require lossless data transmission when the application is configured to manage the process of retransmitting lost packets and correctly arranging received packets. This approach can help to improve the data transfer rate of large files compared with TCP.

2.4 Hyper Text Transfer Protocol (HTTP)

As we are all familiar with the term HTTP and knows about it, this section covers specific details about HTTP that will be needed later to understand the part of feature extraction in chapter 4. The Open Systems Interconnection (OSI[6]) model is a framework that specifies and regulates the functions of a communication system by dividing it into seven logical layers. HTTP is a protocol which is present in the application layer of the OSI model. This layer is the one which is closest to the end user[7]. It is built on top of the Transmission Control Protocol/Internet Protocol (TCP/IP) communication protocol which functions as a request-response protocol in the client-server communication model[8].

Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a web browser or application, are called *requests* and the messages sent by the server as an answer are called responses. HTTP is an extensible protocol which has evolved over time. It is an application layer protocol that is sent over TCP, or over a TLS-encrypted TCP connection, though any reliable transport protocol could theoretically be used. Due to its extensibility, it is used to not only fetch hypertext documents, but also images and videos or to post content to servers, like with HTML form results. HTTP can also be used to fetch parts of documents to update Web pages or an application on demand.

[6] http://en.wikipedia.org/wiki/OSI_model

[7] http://tools.ietf.org/html/rfc2616

[8]http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

2.4.1 HTTP and connections

A connection is controlled at the transport layer, and therefore fundamentally out of scope for HTTP. Though HTTP doesn't require the underlying transport protocol to be connection-based; only requiring it to be reliable, or not loose messages (so at minimum presenting an error). Among the two most common transport protocols on the Internet, TCP is reliable and UDP isn't. HTTP subsequently relies on the TCP standard, which is connection-based, even though a connection is not always required.

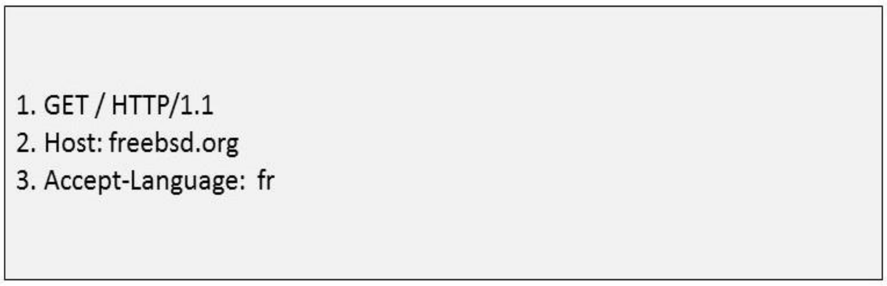
HTTP/1.0 opened a TCP connection for each request/response exchange, introducing two major flaws: opening a connection needs several round-trips of messages and therefore slow, but becomes more efficient when several messages are sent, and regularly sent: *warm* connections are more efficient than cold ones. In order to mitigate these flaws, HTTP/1.1 introduced pipelining and persistent connections: the underlying TCP connection can be partially controlled using the Connection header. HTTP/2 went a step further by multiplexing messages over a single connection, helping keep the connection warm, and more efficient. Experiments are in progress to design a better transport protocol more suited to HTTP.

2.4.2 HTTP flow

When the client wants to communicate with a server, either being the final server or an intermediate proxy, it performs the following steps:

1. Open a TCP connection: The TCP connection will be used to send a request, or several, and receive an answer. The client may open a new connection, reuse an existing connection, or open several TCP connections to the servers.

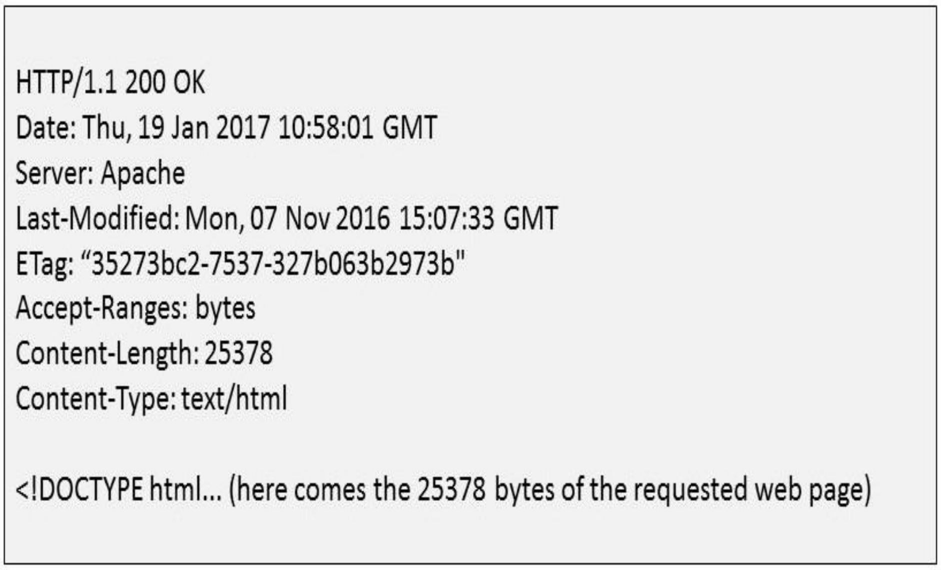
2. Send an HTTP message: HTTP messages (before HTTP/2) are human-readable.it should be noted, in this thesis all of the http flows in our dataset are HTTP/1.1.



```
1. GET / HTTP/1.1
2. Host: freebsd.org
3. Accept-Language:  fr
```

Figure 3: Example of request method Get

3. Read the response sent by the server:



```
HTTP/1.1 200 OK
Date: Thu, 19 Jan 2017 10:58:01 GMT
Server: Apache
Last-Modified: Mon, 07 Nov 2016 15:07:33 GMT
ETag: "35273bc2-7537-327b063b2973b"
Accept-Ranges: bytes
Content-Length: 25378
Content-Type: text/html

<!DOCTYPE html... (here comes the 25378 bytes of the requested web page)
```

Figure 4: Example of response send by server

4. Close or reuse the connection for further requests.

If HTTP pipelining is activated, several requests can be sent without waiting for the first response to be fully received. HTTP pipelining has proven difficult to implement in existing networks, where old pieces of software coexist with modern versions. HTTP pipelining has been supersede in HTTP/2 with more robust multiplexing requests within a frame.

2.4.3 Requests

An example HTTP request:

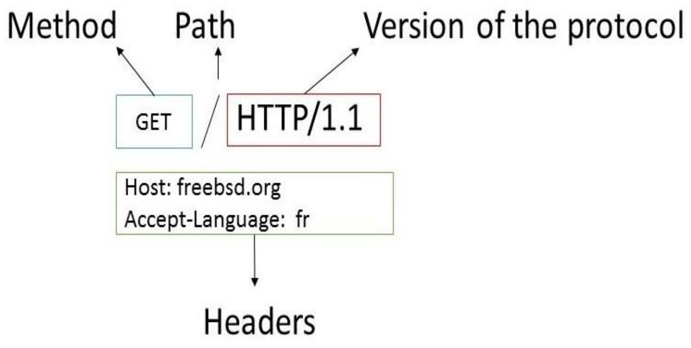


Figure 5: Component of http request

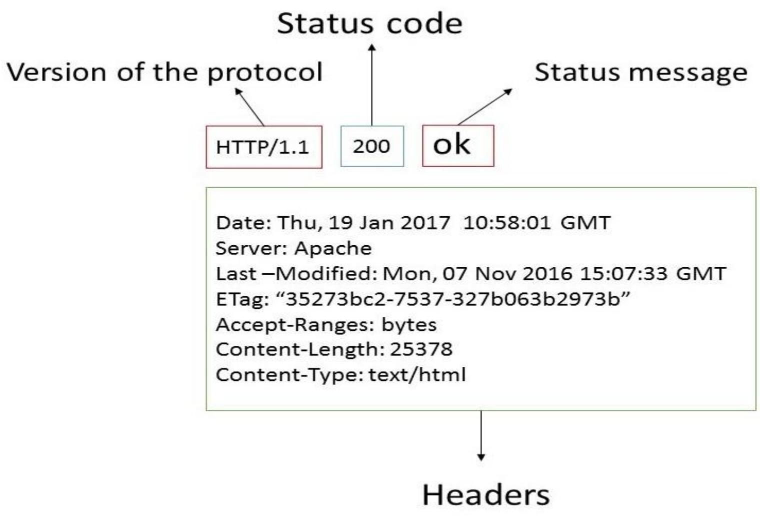
Requests consists of the following elements:

- An HTTP method, usually a verb like GET, POST or a noun like OPTIONS or HEAD that defines the operation the client wants to perform. Typically, a client wants to fetch a resource (using GET) or post the value of an HTML form (using POST), though more operations may be needed in other cases.
- The path of the resource to fetch; the URL of the resource stripped from elements that are obvious from the context, for example without the protocol (http ://), the domain (here freebsd.org), or the TCP port (here 80).

- The version of the HTTP protocol.
- Optional headers that convey additional information for the servers.
- Or a body, for some methods like POST, similar to those in responses, which contain the resource sent.

2.4.4 Responses

An example responses:



```
Date: Thu, 19 Jan 2017  10:58:01 GMT
Server: Apache
Last –Modified: Mon, 07 Nov 2016 15:07:33 GMT
ETag: "35273bc2-7537-327b063b2973b"
Accept-Ranges: bytes
Content-Length: 25378
Content-Type: text/html
```

Figure 6: Component of http response

Responses consist of the following elements:

- The version of the HTTP protocol they follow.
- A status code, indicating if the request has been successful, or not, and why.
- A status message, a non-authoritative short description of the status code.
- HTTP headers, like those for requests.

- Optionally, but more common for requests, a body containing the fetched resource. In any case HTTP is an extensible protocol that is easy to use. The client-server structure, combined with the ability to simply add headers, allows HTTP to advance along with the extended capabilities of the Web or applications.

2.5 Machine Learning

We selected the most common machine learning classification techniques that were used in the literature to detect malwares from benign applications. How works The Machine Learning techniques studied in our work are the following.

Machine learning is a discipline of computer science that concentrates on methods and algorithms that can accomplish or approve certain tasks by utilizing given data. In most cases, this data is needed as input to construct some kind of model, which then can be used to accomplish the task, or generate some information about new, unseen data. There are many examples for machine learning applications. Many artificial opponents for board games use large amounts of data to learn how to play and win the game. This data may contain session reports of games that famous human players have played, but may also be generated from games that the algorithm plays against itself or other algorithms. In many games like chess or backgammon these approaches have created artificial players that can beat even the world's best human players. Another example of machine learning applications can be found in recommender systems of shopping websites. With the vast amount of user data and buying records, many of these sites try to guess what the current user may be interested in, based on his purchase history, content of his shopping cart or currently visited item. The guessed items can then be presented to the user, so he can see and add them to his cart.

The core idea of learning is that knowledge is generated from experience. The generated knowledge differs for every application, and similarly the input used as experience can have many different forms and can be used and presented in many different ways. Machine learning approaches are often differentiated into two major categories:

• Supervised learning

• Unsupervised learning

In supervised learning scenarios, datasets contain pairs of inputs with corresponding outputs. The learning algorithm tries to learn patterns in a training set of input-output (label) pairs, so it can predict the correct output or class for new, unseen input datasets. To compare and evaluate supervised learners, only a portion of the known, labelled data (the training set) is used for training the algorithm. Another portion of the data that has not been used for learning is then used as a test set. The algorithm is used to predict the output class of this data, and the predictions can be compared to the real output to measure the performance of the algorithm. By using the same training and test sets, different methods can be compared with these performance measures. An example for a supervised learning scenario can be seen in spam mail recognition, the e-mails as input data and an output label "spam / no spam" for every e-mail. A system can be trained on many examples that have been categorized, so it can predict if new e-mails are considered spam or not. Unsupervised learning does only contain input data, with no distinct output label. The goal of an unsupervised learning algorithms changes from prediction to pattern recognition, as the datasets are tried to be clustered or categorized. This can be found in image segmentation techniques that try to group all the pixels of one image into similar regions. In following we overview on supervised algorithms used in this thesis.

2.5.1 Decision Tree

Decision tree learning [3] is a category of algorithms to solve supervised machine learning scenarios. Using all the labelled input data, these algorithms try to build a model of one or more decision trees, which are then used to predict the output label of new data examples. Each example is represented by an array of features. Features can be binary, numeric, character, or other data type. These features may be used to split the training data into many parts in order to separate the data of every possible output type. An example of Decision Tree is in Figure 7. In this figure, data consist of different shapes with different properties. Some of them are triangles that may point upwards or not, and all of them can be categorized by their amount of corners. The desired class attribute is the color of the shape that can be either red or blue. By using some decision tree algorithm, several tests on the different shape features have been determined and consecutively connected, to sort the shapes by color. All tests are applied until the bottom of the tree is reached. These leaf nodes contain the class label that was predominantly observed for all or most of the training examples that followed this exact line of tests, so this class label is predicted for every new example with the same nature. In this example, each triangle that is pointing upwards will be predicted to be red.

The goal of decision tree algorithms is to generalize from the different features to the class label. For this reason, it is not desired to split the tree too often until every training example is separated from each other.

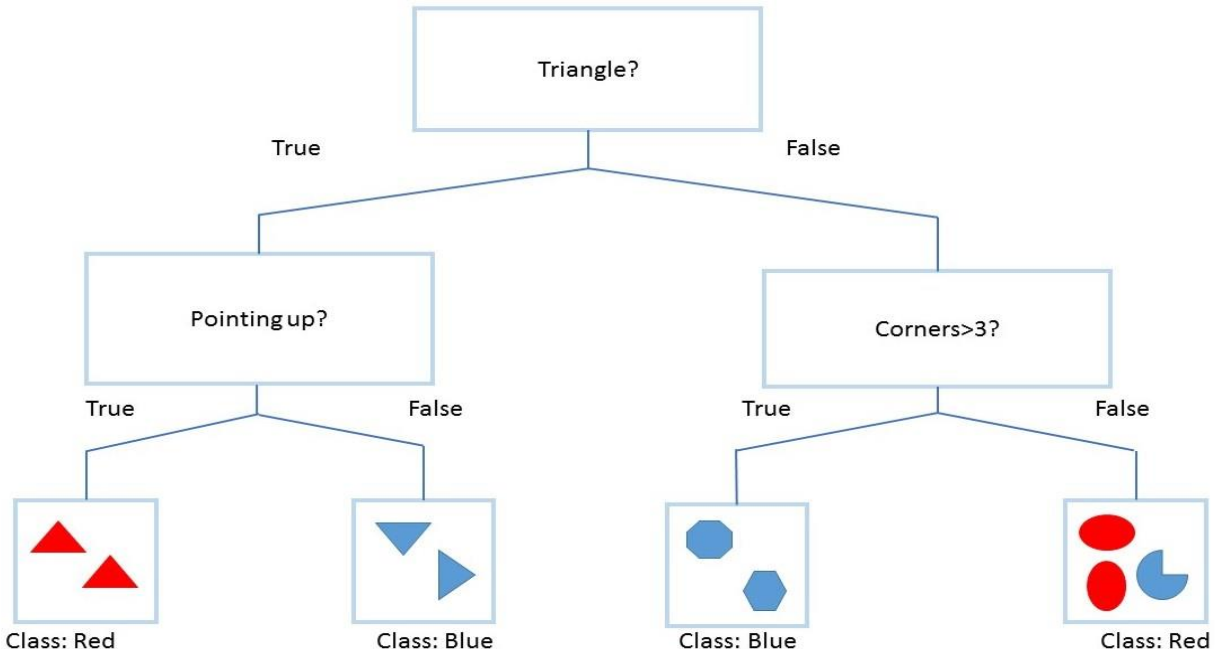


Figure 7: Decision Tree

In this case, every training example would be categorized correctly, but the algorithm would not have generated general rules how to categorize new data, instead it tried too hard to match the specific properties of the training set. This problem is known as overfitting, and it has been shown that simpler trees often display much better performance as complicated trees on test sets, even if the performance of the training set is worse.

2.5.2 Random Forest

Instead of using a single decision tree Random Forest uses many different decision trees for its predictions [4]**.** The number of decision trees **n** is a parameter. For every decision tree, a separate bootstrap [5] sample of the training data instances is created, which is a resampling with replacement. Additionally, at each construction step of the decision trees, the best possible feature is not selected from all remaining features, but from a smaller random subset. The size of this subset is another important degree of freedom when using Random Forest. Each decision tree is constructed separately. To classify a new instance of data, each decision tree gives an individual prediction, and the final prediction is a majority vote of all single predictions. See Figure 8 for an example of this classification process.

The fact that many different decision trees have to be constructed may suggest that the runtime of Random Forest can be a big problem, but in fact, this algorithms proves to be very fast. Each decision tree only gets a small subsets of instances that have to be considered, and the amount of features that have to be examined at each construction step is also much smaller. With this combination, each decision tree can be constructed fast and since the construction of every tree is completely separate, this process can be parallelized for higher efficiency. This is also true for the classification process of new instances, as each single prediction can be made individually. The majority vote at the end of the classification process is simple.

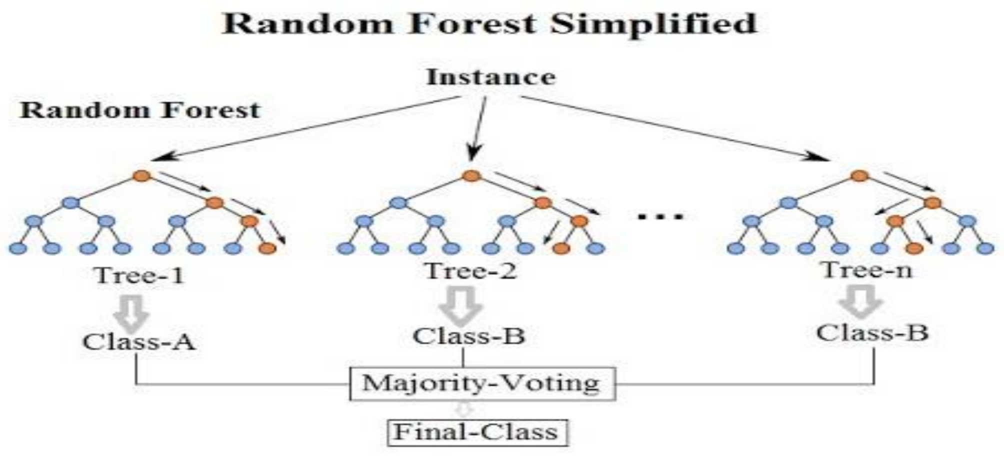


Figure 8: Random Forest algorithm

2.5.3 Naive Bayes

The Bayesian classifiers [6] are linear classifiers efficient and relatively simple to implement. It takes advantage of the fact that often the relationship between the values of feature and the class is not deterministic. Therefore, to better describe this relationship, it relies on a probabilistic model based on Bayes' theorem. In particular, during the course of this thesis we are using naive Bayesian classifiers, it is believed that the attributes of the elements considered are independent of each other, unrealistic assumptions but generally proves effective in practical problems. There are more types of Bayesian classifiers and are distinguished according to the type of distribution of probability that use; very common hypothesis is that the features have a Gaussian probability distribution, so the Gaussian Naive Bayes classifier is widespread. As it can be seen in chapter 5, in this study the best results are obtained instead assuming that the feature in question to have a multinomial probability distribution.

2.5.4 Linear Discriminant Analysis

Fisher (1936) proposed linear discriminant analysis (LDA) [7] as a method for classifying observations or objects into one of two mutually exclusive and exhaustive groups based on a linear function of a set of independent variables associated with each observation or object. The linear function of LDA is chosen to maximize a group separation metric. In calculating this linear function, the important variables should be identified and the function can then be used to allow new observations to be classified as belonging to one of the predetermined groups (e.g. Orgler, 1975) Consider a two-group discriminant problem in which n features are associated with each observation, with x=($x_1, x_2, \ldots, x_n$) representing the vector of feature values. The objective of LDA is to estimate P(y|x), the probability of membership of group y, y=1,2 , given feature vector x. It is assumed that the covariance matrices for each group are equal. An observation with vector of feature values x is then classified by considering the function w'x=c , where c is a cutoff value, such that if w'xc the observation is classified as belonging to group 2. In general, the cutoff value, c, will depend on the prior probabilities of group membership and the costs of misclassifying observations in each group (e.g. Hand, 1997). The assumption of equal covariance matrices can be relaxed and a quadratic discriminant function generated (e.g. Smith, 1947)

-------- Linear Discriminant Analysis

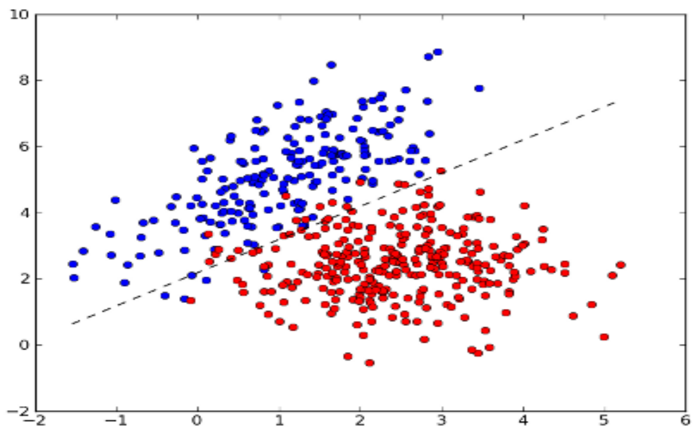


Figure 9: Linear Discriminant Analysis

2.5.5 Nearest Neighbors

Nearest neighbor methods, such as the k-nearest neighbor (k-NN) method, are popular algorithms used in machine learning [7]. The idea is that new data points will be classified based on what the majority of the nearest neighbor points have been labeled as in the training data. An example of this can be seen in Figure 10. The amount of neighbor points being evaluated is determined by the input variable k, which usually is an odd number to avoid cases where a new data point is surrounded by an equal distribution of two classes. For some situations, it could be beneficial to weigh the "vote" for the neighbors depending on their distance so that closer points will contribute more to classification than distant ones even though they still are neighbors. One should also keep in mind that the efficiency of k is highly dependent on the data set being analyzed. The classification algorithm will benefit from a high k value if the data is noisy, but could also be exposed to overfitting due to this. The K-NN is also lazy learning algorithm that means it does not create a generalized model during the learning phase, as most of the other classifiers. Instead it relies on having access to the whole training dataset when performing classification on the new data. Another thing is that the algorithm does not make any assumptions of the data distribution which could come in handy when analyzing manually data from real world applications. The drawback of being extremely fast in the training phase is that this lazy algorithm becomes computationally heavy when performing the actual classification of the data. It also requires a lot of memory to process if the training set is large since it needs direct access to all the data.

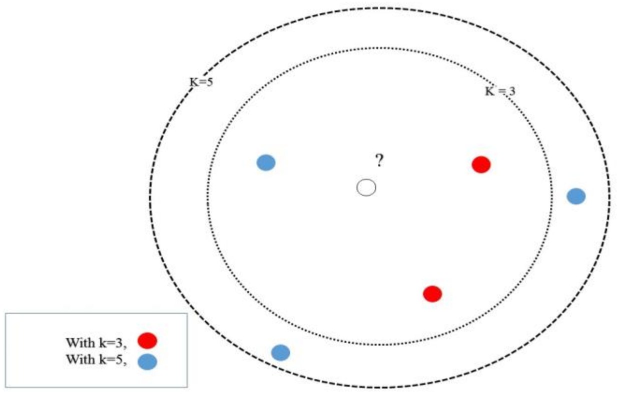


Figure 10: K-NN algorithm

2.5.6 Support Vector Machine

The Support Vector Machines (SVM) [7] are a set of supervised learning methods for solving problems of classification or regression. It is a binary classification method, not a probabilistic one. Its purpose is the search on hyperplane with optimal separation between the two possible classes in the space of features. Also it is possible to extend the operation to problems in which the number of classes to predict is greater than two. In the case where the input data are not linearly separable, one can use the functions calls kernel which map the initial data in a higher dimensional space, where it is possible find a separable hyperplane. SVM are therefore very powerful classifiers that allow attributable to the linear classification even when one has to solve very complex problems. For the purposes of this work is not necessary, however, a strict mathematical analysis of the support vector machines, so will just give you an example for clearer their operation of SVM in a case of linear binary classification: As it can be seen in Fig. 13, in this simple example it is the line h to divide the two classes with the maximum margin separations.

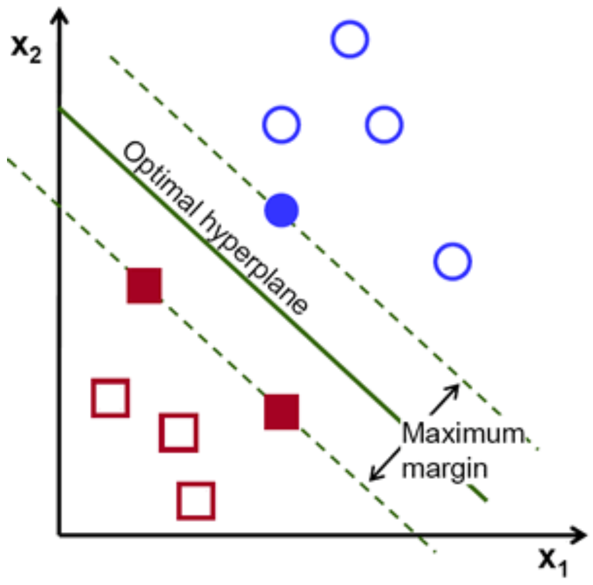


Figure 11: Support Vector Machines classifier

2.5.7 AdaBoost

AdaBoost [8], short for Adaptive Boosting, is a meta-algorithm, and can be used in conjunction with many other learning algorithms to improve their performance. AdaBoost is adaptive in the sense that subsequent classifiers built are tweaked in favor of those instances misclassified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems, however, it can be less susceptible to the overfitting problem than most learning algorithms. The classifiers it uses can be weak (i.e., display a substantial error rate), but as long as their performance is not random (resulting in an error rate of 0.5 for binary classification), they will improve the final model.

AdaBoost generates and calls a new weak classifier in each of a series of rounds *t = 1… T*. For each call, a distribution of weights *D (t)* is updated that indicates the importance of examples in the data set for the classification. On each round, the weights of each incorrectly classified example are increased, and the weights of each correctly classified example are decreased, so the new classifier focuses on the examples which have so far eluded correct classification.

Ensemble methods [5] (in case of this algorithm boosting methods) use multiple models to obtain better predictive performance than could be obtained from any of the constituent models. In other words, an ensemble is a technique for combining many weak learners like weak classifier 1, 2, 3 in Figure 12 to in an attempt to produce a strong learner with combination of the weak classifiers. Evaluating the prediction of an ensemble typically requires more computation than evaluating the prediction of a single model, so ensembles may be thought of as a way to compensate for poor learning algorithms by performing a lot of extra computation.

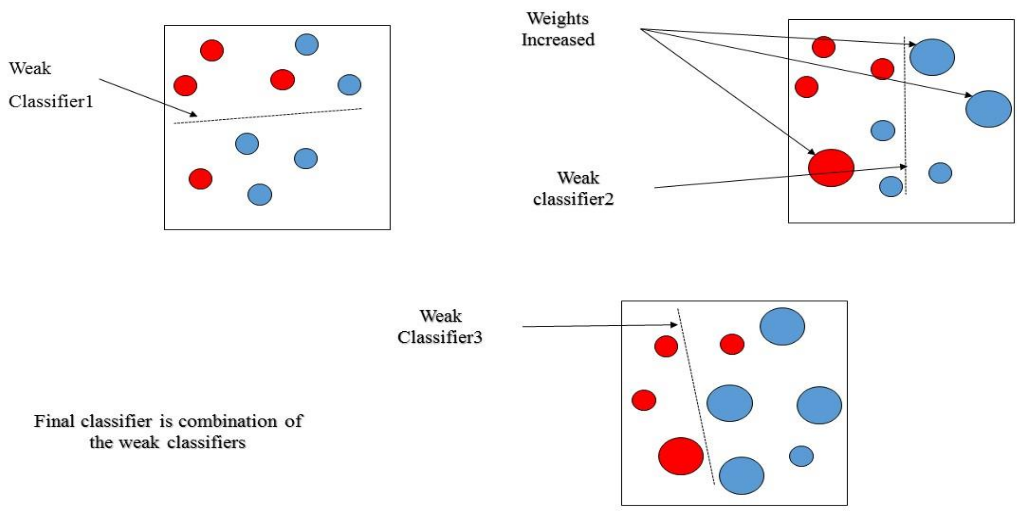


Figure 12: Implementation steps of AdaBoost classifier

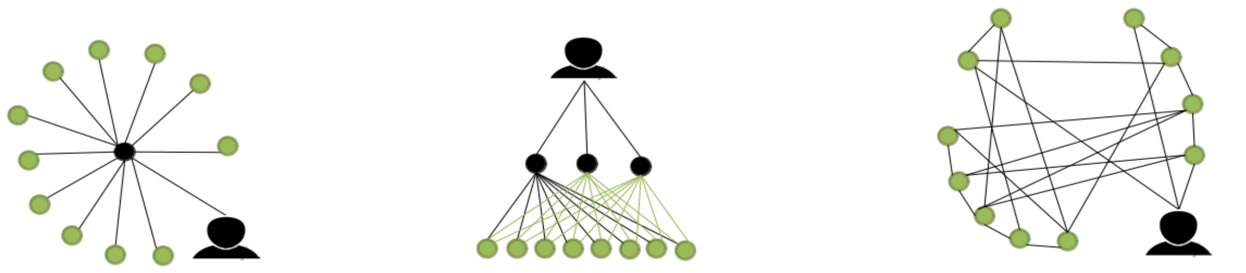
2.6 Botnet malware definition

Botnets define as malicious software or application that installed into the victim machine and sends information without the consent of the owner. The first part of botnet features is, the Bots usually achieve control of the target's device without having the attacker to log in the target's device. In fact the bots communicate with each other using the C&C server to receive an instruction from the attacker to achieve the same goal [9]. The main challenge with a botnet server is that one or more servers could be linked to each other to control a few hundreds if not thousands of bots client that called zombies.

The second part of the botnet features is the botnets software features. One feature is that the attacker cannot be reachable meaning that the attacker is hidden by using many IP addresses flooding a single target [10]. This feature does not only hide the identity of the attacker it also hides the way that the attacker comes to the target's machine [10]. The bots receives the instruction from the botnet master through the C&C control channel, and then the bots will perform the task that the botnet master asked for and report the result through the C&C control channel. This means that the bots are able to adapt with any environment as well as being accurate and targetable [11]. Botnets classified into two phases that required being consider.

2.7 Botnet architectures

When bots organize in groups that can be remotely controlled by attackers, we term the originating network a botnet. As with normal networks, botnets are also structured in various architectures. One inherent property of all botnet architectures is that the network allows a botmaster to send commands to the bots in some way. Similarly, although not a strict requirement in every botnet, most botnet designs also allow the bots to send feedback to the botmaster. But apart from that, botnet architectures predefine unique characteristics that are important when analyzing botnet.



A) Centralized B) Semi-Distributed C) Peer-to-Peer

Figure 13: Comparison of botnet architectures

(Green nodes represent bots, dark nodes represent c&c servers and person symbols represent botmasters)

As visible in Figure 13 shows the three botnet architectures commonly used by botmasters: Centralized, Semi-distributed and Peer-to-peer (P2P) botnets. We will describe these three architectures in the following subsections.

2.7.1 Centralized botnets

In centralized botnets, as shown in Figure 13.A, all bots connect to a single C&C server. From an attacker's perspective, this client-server architecture is trivial to implement. The endpoint address of the C&C server is usually a hard-coded string within malware binaries, and bots communicate with the servers over IRC, HTTP or proprietary protocols. The main resilience bottleneck of a centralized botnet is it centralized server. If defenders take control over or disrupt the C&C server, the clients can neither request commands, nor send feedback to the servers anymore. Defenders often aim to sinkhole a botnet, in that they deploy infrastructures that mimic the existing C&C endpoints, for example, by redirecting a C&C domain to a server controlled by the defender (the sinkhole). When sinkholing a C&C endpoint, the defenders do not serve commands to the bots, rendering them useless for the botmasters. At the same time, the defenders can identify infected systems by looking at the sinkhole log files.

2.7.2 Semi distributed botnets

Over the years, attackers noticed the drawbacks of centralized botnets and developed various strategies to distribute C&C servers. The goal was to mitigate the single point of failure of centralized botnets, while retaining the client-server model of centralized botnets. Figure 13.B shows that in semi-distributed botnets, the bots contact multiple different C&C servers.

The semi-distributed design significantly improved botnet resilience and thus raised the bar for botnet takedowns. Botmasters invented different schemes to achieve a higher degree of distribution. Most trivial, bot binaries contain multiple server addresses, and in case one C&C is not reachable anymore, the bot communicates with another. A special and more advanced kind of such distribution is achieved by using domain name generation algorithms (DGAs). DGAs precompute C&C server hostnames by algorithms that accept deterministic seeds (such as the current date). The botmaster knows the DGA and needs to host only the C&C server under one of the generated domains. A defender would need to register up to hundreds of frequently changing C&C domains to fully disrupt the botnet.

Attackers use DNS fast-flux networks as a complementary technique to increase botnet resilience. In fast-flux, the set of C&C server IP addresses is rapidly changed via DNS by either the bots or the botmaster. As opposed to DGAs, if the botmaster controls the IP address fluctuation, defenders cannot compute the future end points of C&C servers. In double fast-flux botnets, also the C&C domain authoritative name servers are served round-robin, adding another layer of fluctuation.

Summarizing, semi-distributed botnets are significantly more resilient than centralized botnets, although they still have (redundant) centralized servers.

2.7.3 Peer-to-peer botnets

Peer-to-peer (P2P) botnets are fully distributed botnets, in which the bots retrieve their commands from other bots via the P2P network. As Figure 13.C shows, P2P bots keep track of other bots in the botnet, following an architecture without central servers. The lack of central components makes the resilience of P2P architectures attractive for botmasters. In particular, the botnet continues to operate even if a large number of bots are removed from it, and the P2P network quickly heals itself from sudden network changes. On the other hand, P2P networks are prone to other mitigation techniques, ranging from enumerating all infected bots to P2P-based sinkholing or P2P network partitioning.

Chapter 3

Related work

Scientific publications dealing with the problem of classification of malwares in the Android area are numerous. They are analyzed following only the most relevant and promising for the objective of this thesis.

3.1 Dynamic analysis

As recently proposed by Antivirus companies, static analysis can be deployed for malware detection in Android devices. But due to the limited resources of smartphones, most of the recent proposals for malware detection on Android devices are based on behavior analysis for anomaly detection. Dynamic analysis examines the application during execution. It may miss some of the code sections that are not executed but it can easily identify the malicious behaviors that are not detected by static analysis methods. Although static analysis methods are faster to malware detection but they fail against the code obfuscation and encryption malwares.

3.1.1 Anomaly based detection

In [12] proposed CrowDroid to detect the behavior of applications dynamically. Details of system calls invoked by the app are collected by the Strace tool [13] and then crowdsourcing app, which is installed on the device, creates a log file and sends it to remote server. Log file may include the following information: Device information, apps installed on device and system calls. K-mean clustering algorithm is applied at server side to classify the application as malware or benign.

In [14] proposed Andromaly, a behavioral based Android malware detection system. In order to classify the application as benign or malware it continuously monitor the different features and patterns that indicate the device state such as battery level, CPU consumption etc. While it is running and then apply the machine learning algorithms to discriminate between malicious and benign apps.

Anti MalDroid [15], a malware detection framework using SVM algorithm is proposed by Zhao, can identify the malicious apps and their variants during execution. First it monitors the behavior of applications and their characteristics then it categorize these characteristics as normal and malicious behavior.

In [16] performed an analysis of Android botnets that employ HTTP traffic for their communications. By clustering the generated network traffic of different Android malware with the usage of an algorithm originally developed for grouping desktop malware, they showed that the samples belonging to the same malware family have similar HTTP traffic statistics. Moreover, a small number of signatures can be extracted from the clusters, allowing to achieve a good tradeoff between the detection rate and the false positive rate. Since this work contained of the large malware dataset we use their malware dataset in our thesis.

In [17] a traffic analysis is done for detecting the malwares on Android smart phones based upon their network traffic features. This is a work done for Android malware detection based on analyzing the network traffic features. This approach is not general for all the malwares but specific only to those set of malwares which connect to some remote server in the background and hence produce some network traffic. Android malware samples used in the experiments are 45 sample that received from Android Malware Genome Project of North Carolina State University. The accuracy reported with selected rule based classifier is about 93%.

3.1.2 Emulation based detection

Yan et al. [18] present Android dynamic analysis platform DroidScope, based on Virtual Machine Introspection. As the anti-malware detect the presence of malwares because both of them reside in the same execution environment so the malwares also can detect the presence of anti-malware. DroidScope monitors the whole operating system by staying out of the execution environment and thus have more privileges than the malware programs. It also monitors the Dalvik semantics thus the privilege escalation attacks on kernel can also be detected. It is built upon QEMU. DroidDream and DroidKungFu [19] were detected with this technique.

Blaising et al. [20] proposed Android Application Sandbox (AASandbox) which detect the suspicious applications by performing both static and dynamic analysis on them. It first extracts the .dex file into human readable form and then performs static analysis on application. Then it analyzes the low level interactions with system by execution of application in isolated sandbox environment. Actions of application are limited to sandbox due to security policy and do not affect the data on device. It uses Money tool to dynamically analyze the application behavior which randomly generates the user events like touches, clicks and gestures etc. it cannot detect the new malware types.

3.2 Static analysis

In static analysis, the features are extracted from the application file without executing the application. This methodology is resource and time efficient as the application is not executed. But at the same time, this analysis suffers from code obfuscation techniques the Malware authors employ to evade from static detection techniques. One of very popular evasion technique is the Update Attack: a benign application is installed on the mobile device and when the application gets an update, the malicious content is downloaded and installed as part of the update. This cannot be detected by static analysis techniques which will scan only the benign application.

3.2.1 Permission based analysis

In [21] is proposed DREBIN, an efficient method for the recognition of malware by static analysis. Starting from file AndroidManifest.xml and from decompiled code of apps they are extracted sets of features submitted to a support vector machine classifier (SVM). This method is tested on a set of 123,453 benign applications and 5560 malwares. Obtaining results that indicate an accuracy 94%, with only 1, 1%false positive. This methodology can also be used directly on mobile devices, with execution times in the order of a few seconds. For each app examined, DREBIN also provides a description of the identified suspicious features. The malware collection used in [21] is made available as dataset DREBIN.

The authors of [22] developed a methodology based not only on the declaration of permission from Android app, but also on those actually used. Furthermore, for the creation of the feature vectors, they are not only considered the individual permission, but also the pairs of permissions (both declared is actually used).The proposed technique is divided into two sequential phases, initially controlling the required permissions and then the pairs of permissions actually used. Empirically excellent results are obtained, using as classifier decision tree and conducting tests on a set of benign 28,548 applications and 1,536 malware.

In [23] it is presented a similar to [22] work, as techniques are used to Machine learning of the feature sets extracted from file AndroidManifest.xml of Android applications (especially the permissions declared) With a dataset consisting of 249 malware and 357 benign applications, in [23] are evaluated the performance of different classifiers to identify malicious apps including decision trees, Random Forest e Naive Bayes.

The authors of [38] suggest a machine learning based on the methodology for recognize the malwares, using k-means and decision trees algorithm. From each analyzed application is extracted a feature vector formed by 0 and 1, mainly affecting the required permissions from the app, where 1 indicates the specific request permission and 0 the absence of request permission The method developed in [38] is validated with dataset of 500 applications and obtaining good results.

3.2.2 Signature based approach

Signature based malware detection methods are commonly used by commercial anti malware products. This method extracts the semantic patterns and creates a unique signature [39]. A program is classified as a malware if its signature matches with existing malware families‟ signatures. The major drawback of signature based detection is that it can be easily circumvented by code obfuscation because it can only identify the existing malwares and fails against the unseen variants of malwares. It needs immediate update of malware variants as they are detected.

In [24] proposed AndroSimilar, a robust statistical signature method to detect the unknown variants of existing malwares that are usually generated by using repackaging and code obfuscation techniques. It generates the variable length signature for the application under test and compares it with the signatures in AndroSimilar malware database and identify the app as malware and benign on the basis of similarity percentage.

DroidAnalytics [25] is a signature based analytic system which extract and analyze the apps at op-code[9] level. It not only generates the signature but also associate the malware with existing malwares after identifying the malicious content. It generates 3 level signatures. First it generates signature at method level by API call tracing then combining all the signatures of methods in a class it generates the class level signatures and at third level it generates the application signature by combining the signatures of the classes in the application.

[9] https://en.wikipedia.org/wiki/Opcode

Chapter 4

Methodology and Design

In this chapter we describe how we implemented our framework for the dynamic analysis, detection and classification of Android malware. The implementation notes are divided in two parts:

A. **Botnet malware detection;** to achieve this aim we follow three steps (see Figure 14). The first step is Data collection that contained two dataset: first dataset is the benign dataset was created by self since there was no standard dataset available and second dataset is malware dataset that we took them from the authors of [16] where they gather a dataset from Android botnets. After data collection, in the next step the obtained features are parsed from the network traffic captured of a large sample of malware and benign apps; We propose two separate features selection and extraction approaches that leads to various result using machine learning algorithms and in chapter 5 compare them results. Third step consists of building classifiers upon above found distinguishing features and running the classifier on the test data to prove its accuracy.

B. **Botnet family classification;** in the second part of this chapter we detect families of botnets malware. Our malware dataset contains 23 different botnets families. Similarly to (A) we propose two approaches and finally the compare obtained results. We use supervised machine learning approaches for the detection of botnets families. The method applied is the same as (A) with the difference that in the first feature extraction method we focus on analyzing the specific statistical information and in the second method on structural similarities among malicious HTTP traffic traces generated by executing botnets malware.

To implement the proposed methods we use tshark [26] that is a network protocol analyzer to manipulate network traffic logs together with python programming language.

4.1 Workflow for malware detection

Figure 14 illustrates the details of the workflow for malware detection. There are three main phases: data collection (see Section 4.2), feature selection and extraction (see Section 4.3), and machine learning classifiers (see Section 4.4).

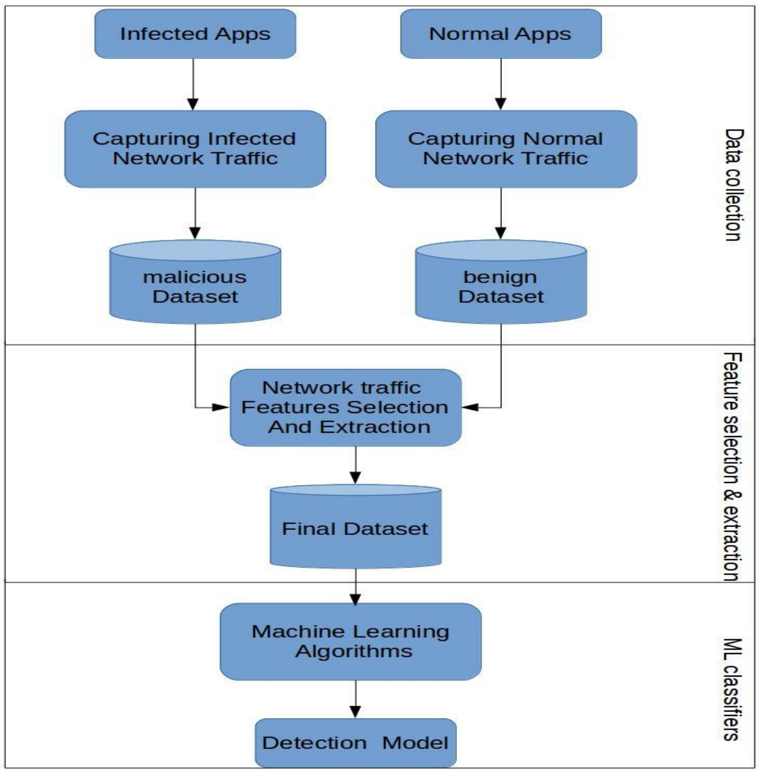

Infected Apps
Normal Apps
Capturing Infected Network Traffic
Capturing Normal Network Traffic
malicious Dataset
benign Dataset
Data collection
Network traffic Features Selection And Extraction
Final Dataset
Feature selection & extraction
Machine Learning Algorithms
Detection Model
ML classifiers

Figure 14: Workflow for malware detection

4.2 Data collection

To apply machine learning algorithms, it is first necessary to build a dataset that contains a large number of samples to be analyzed. This dataset mainly consists of two part: the first the malware dataset that is explained in Section 4.2.1 and the second is the benign dataset that is explained in Section 4.2.2.

4.2.1 Malware dataset

For the analysis of malware network traffics we used a dataset created by Marco Aresu in [16]. He considered 817 malware applications from 23 different malware families to evaluate the effectiveness of the classification procedure. The samples of [16] were gathered from Malgenome [27], Contagio [28], Drebin [21] and VirusTotal [29]. Each Malware sample is executed in a controlled environment like Anubis [30] and their network traffics is recorded in the file with pcap format. In this way they succeeded to gather 814 files pcap that constitute our malware dataset. Because of the heterogeneous nature of the sources, each file pcap is associated with a unique MD5 hash to avoid having duplicate.

The 23 family used in our dataset, 11 samples belongs to AndroRat, 3 samples to AVPass, 3 samples to BlackFlash, 3 samples to BadNews, 112 samples to BaseBridge, 45 samples for BgServ, 3 samples to Chulli, 86 samples to DroidKungFu, 69 sample to Extension, 151 sample to FakeAngry, 8 samples to FakePlay, 11 samples to FakeTimer, 106 samples to Fjcon, 23 samples to Geinimi, 3 samples to GoldenEagle, 6 samples to Lien, 3 samples to NickiSpy, 3 sample to Obad , 119 samples to plankton, 25 samples to RootSmart, 3 samples to Skullkey, 5 samples to SMSpacem and finally 13 samples to Tracer. A list of the malwares families and number of malwares used in our experiments is given in Table 1. The malware family column refers to the name of the malware variant, and the related number shows the considered malware samples for that variant.

Malware family	**#samples**	**Malware family**	**#samples**
AndroRat	11	Fjcon	106
AVPass	3	Geinimi	23
BackFlash	3	GoldenEagle	3
BadNews	3	Lien	6
BaseBridge	112	NickiSpy	3
BgServ	45	Obad	3
Chulli	3	Plankton	119
DroidKungFu	86	RootSmart	25
Extension	69	Skullkey	3
FakeAngry	151	SMSpacem	5
FakePlay	8	Tracer	13
FakeTimer	11		
		Total	814

Table 1: Malware dataset

4.2.2 Benign dataset

To evaluate the quality of the considered methods and their implementations, it is first necessary to build a dataset that contains a large number of benign samples like malware dataset to be analyzed. Google play store is the official channel for Android applications, these are considered (initially only) not malicious apps. In google play Store there exist 36 applications categories; we created a crawler in python to download .apk file for 15 most downloaded applications in each category of google play store. In this way we gathered 540 applications from the play store. To be mixed our normal dataset with various types of applications not only the most downloaded applications we have chosen 8 applications with rate of review less than 2 in each category of google paly store ;In this way we addition 288 applications to 540 Previous applications. So already gather 828 normal applications from the google play store.

Then, we execute each application in a controlled environment, and capture their network traffic. We provided Galaxy Tab 3 as device, this platform provides a basic Android simulation environment and command line mode of interaction. Our model is designed to install and activate applications to generate traffic traces automatically, which consists of three phases: automatic installation, app stimulation and traffic collector.

First implements automatic installation and activation of the application using command line Android shell; for app stimulation we use Monkey [31] to simulate real user behavior. Monkey is a tool that runs on device and generates pseudo-random streams of user events such as clicks, touches, gestures, as well as a number of system-level events. The traffic collector is tcpdump [32], a tool designed to capture the inbound and outbound traffic, automatically.

Our malware dataset is populated with pcap files. Each pcap file contains the traffic captured during a stimulation phase. A stimulation takes from 30 second to 5 minutes. Figure 15 depicts the procedure described above.

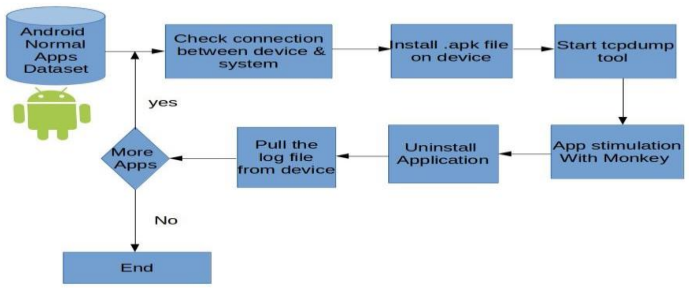

Android Normal Apps Dataset
Check connection between device & system
Install .apk file on device
Start tcpdump tool
App stimulation With Monkey
Uninstall Application
Pull the log file from device
More Apps
yes
No
End

Figure 15: Flow chart for create normal dataset

4.3 Network traffic feature extraction

When there is abundant input data to an algorithm and tend to be more redundant and irrelevant, feature extraction is performed. It is required to gain the precise measurement of features (called feature selection) which influence the classification of input as benign or malicious. The outcome of the feature extraction phase is a vector containing the frequencies of features extracted. Features extracted are chosen such that it attains maximum classification accuracy. The time required to get features from input dataset is also depends on the feature extraction methods. Feature extraction method affects the performance of the system in terms efficiency, robustness, and accuracy.

Since our work is dynamic behavior analysis so we analysis characteristic of network traffic in order to distinguish benign from malicious traffic. The network traffic is a data stream, which means the signature of the traffic will most likely change over time. Moreover, bots tend to change their behaviors over time as results of commands from the bot master or new botnet version replacing the old one. This will create a concept-drift problem [33] for the detection framework. Unlike legitimate network applications, new botnets can spread over the Internet at any time. This means that the detection model may not capture the characteristics of these new bots during the training phase.

4.3.1 Network traffic raw features

Aim of presentation these feature extraction method is that, effort to select features at network level in order to isolate botnet behavior from normal network behavior. We call raw features because these features are base of features proposed in section 4.3.2.

Behaviors-based network traffic identification usually aims at studying different characteristics that occur in network traffic. For example, the packet size, the flow duration, the number of ACK and SYN packets per flow, etc. Behaviors- based identification methods use a combination of these characteristics to distinguish between different types of network traffic. In our model, we extract features about the number of packets incoming per flow, number of packets outgoing per flow, number of bytes outgoing per flow and duration of flow**.** These features has been extracted from TCP and UDP flows (see Table 2). Our feature vector represents the traffic behavior corresponding to a single TCP, UDP flow that labeled as malware or normal traffic.

Number	Features selected and extracted
1	Number of packets In per (TCP,UDP)flow
2	Number of bytes In per (TCP,UDP) flow
3	Number of packets out per (TCP,UDP) flow
4	Number of Bytes out per (TCP,UDP) flow
5	Duration of (TCP,UDP) flow

Table 2: Network traffic raw features

4.3.2 Network traffic statistical features

In this section le feature are constructed in the similar way as down in [17]. Our proposed features in this section can be grouped into three categories, namely, TCP flows-based, UDP flows-based and variance-based features. Summary of These features has been shown in Table 3.

A) The first category involves a set of features that can be extracted from network TCP flows, you can see these features in Table 3 in column Category with label (A). Average packet size, average byte incoming to outgoing ratio, average packet send per flow, average packet receive per flow, average byte receive per flow, average byte send per flow, average packet incoming to outgoing ratio, average byte send per second, average byte receive per second. In our method since each pcap file represents a single execution of Android application we first extract the TCP flows from the captured traffic and save them in a comma separated values (CSV) file and then we calculate features from their column values using arithmetical average calculations.

B) The second category involves a set of features that can be extracted from UDP flows. The procedure used to extract these features represented to (A), but applies to UDP flows. In Table 3 you can see these features in column category with label (B).

C) The third category of features is variance based features that in Table 3, column category seen with label (C). In this type of features, we calculated variance features obtained from parts (A) and (B). Variance is a measurement of the spread between numbers in a data set. The variance measures how far each number in the set is from the mean.

These features are used to distinguish botnet traffic from normal traffic. Feature vectors represents the traffic behavior corresponding to a single application (pcap file) that were stored as a sequence of comma separated values (CSV) files. We labeled to the feature vectors as botnets or normal applications.

Number	Category	Features
1	A	TCP Average Packet Size
2	A	TCP Average Byte Incoming to Outgoing Ratio
3	A	TCP Average packet Send Per Flow
4	A	TCP Average packet Receive Per Flow
5	A	TCP Average Byte Receive Per Flow
6	A	TCP Average Byte Send Per Flow
7	A	TCP Average Packet Incoming to Outgoing Ratio
8	A	TCP Average Byte Send Per Second
9	A	TCP Average Byte Receive Per Second
10	B	UDP Average Packet Size
11	B	UDP Average Byte Incoming to Outgoing Ratio
12	B	UDP Average packet Send Per Flow
13	B	UDP Average packet Receive Per Flow
14	B	UDP Average Byte Send Per Flow

15	B	UDP Average Byte Received Per Flow
16	B	UDP Average Packet Incoming to Outgoing Ratio
17	B	UDP Average Byte Send Per Second
18	B	UDP Average Byte Receive Per Second
19	C	TCP Variance Packet Size
20	C	TCP Variance Byte Incoming to Outgoing Ratio
21	C	TCP Variance packet Send Per Flow
22	C	TCP Variance Packet Receive Per Flow
23	C	TCP Variance Byte Receive Per Flow
24	C	TCP Variance Byte send Per Flow
25	C	TCP Variance Packet Incoming to Outgoing Ratio
26	C	TCP Variance Byte Send Per Second
27	C	TCP Variance Byte Receive per Second
28	C	UDP Variance packet Size
29	C	UDP Variance Byte Incoming to Outgoing Ratio

30	C	UDP Variance packet Send Per Flow
31	C	UDP Variance Packet Receive Per Flow
32	C	UDP Variance Byte send Per Flow
33	C	UDP Variance Byte Received Per Flow
34	C	UDP Variance Packet Incoming to Outgoing Ratio
35	C	UDP Variance Byte Send Per Second
36	C	UDP Variance Byte Receive per Second

Table 3: Network traffic statistical features

4.4 Machine learning algorithms for malware detection

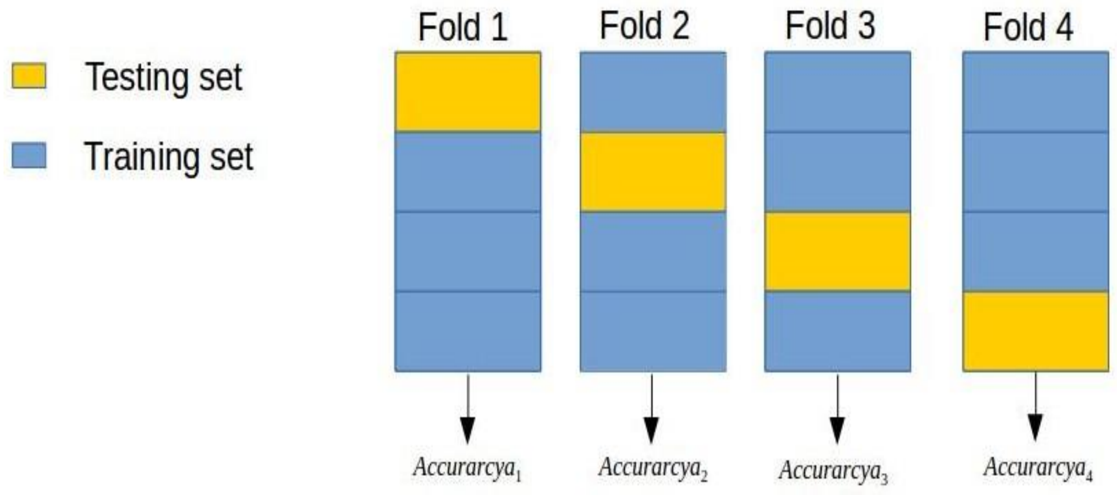
We investigate in this work the ability of different machine learning techniques in addressing the above requirements. We selected the most common machine learning classification techniques that were used in the literature to detect botnet. We used these machine learning classification techniques to investigate the possibility of detecting botnets traffic by analyzing network traffic behaviors only. The ML techniques studied in our work are the following:

- Random Forest classifier
- AdaBoost classifier
- Nearest Neighbors classifier
- Decision Tree classifier
- Support Vector Machine
- Linear Discriminant Analysis classifier

- Gaussian Naive Bayes Classifier
- Multinomial Naive Bayes classifier

4.4.1 Cross validation

This study conducted 10-fold cross-validation [34] to evaluate the performance of machine learning classifiers for the collected dataset. The 10-fold cross-validation first divides dataset in 10 folds and uses the 1st fold as the test set in step 1, while using the remaining folds as train set. In step 2, the 2nd fold is used as the test set while the remaining folds are used as the training set for the total of 10 steps of cross-validation to perform the evaluation. In existing study [14], 80% of the entire dataset has used as training set to evaluate the performance of malware detection, while using the remaining 20% as test set. However, cross validation can more precisely examine than method proposed in [14] as it performs performance evaluation**.** In machine learning, cross validation is known to provide a very good estimation of the generalization error of a classifier. Figure 16 shows an example k-fold cross validation where k = 4.



$$\text{Accuracy}=\frac{1}{k}\sum_{i=1}^{k} Accuracya_i$$

Figure 16: Graphic example for 4-fold cross validation

4.5 Malware family classification

In Section 4.1 we present a workflow for botnets malware detection, in this section we want a step toward and this time propose workflow for classification of botnets families.

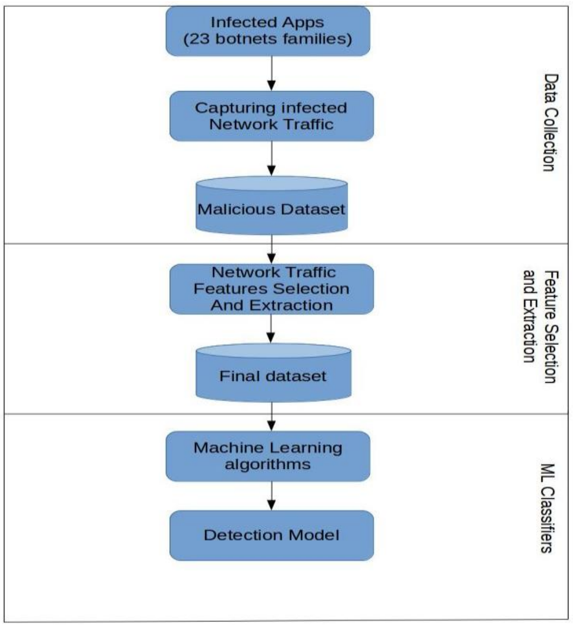

Infected Apps (23 botnets families)
Capturing infected Network Traffic
Malicious Dataset
Network Traffic Features Selection And Extraction
Final dataset
Machine Learning algorithms
Detection Model
Data Collection
Feature Selection and Extraction
ML Classifiers

Figure 17: Workflow for botnets family detection

The most common communication channels used by attackers to control the infected machines are based on the HTTP protocol [35]; In particular more than 99% of Android botnets use the HTTP-based traffics to receive bot commands from their C&C servers [35]. In addition 70% of the generated network traffic by Android apps is spread through this protocol [36]. In this part of thesis. We show how it is possible, classify mobile botnets families by analyzing the HTTP traffic they generate. To do so, we propose two difference method in first method we create malware classes by looking at specific statistical information that are related to the HTTP traces and in second method create classes by looking at structural similarity among of HTTP traces.

4.5.1 Classification based on statistical features

In this section, the features are built in a way similar to [37]. It leverages seven statistical features to achieve the classification of Android malware into families.

The goal of feature selection and extraction phase is to find simple statistical similarities in the way different malware samples interact with the remote server. Let $M = \{m^{(i)}\}\ i = 1 \dots N$ be a set of malware samples, and $H(m^{(i)})$ be the HTTP traffic trace obtained by executing $m^{(i)} \in M$ (for fixed time T). We translate, each trace $H\left(m^{(i)}\right)$ into a pattern vector $v^{(i)}$containing the following statistical features: (i) the total number of HTTP requests, (ii) the number of GET requests, (iii) the number of POST requests, (iv) the average length of the URLs, (v) the average number of parameters in the request, (vi) the average amount of data sent by POST requests and (vii) the average length of the response (see Table 4).

FEATURE EXTRACTED FROM HTTP REQUESTS	
1	Total number of HTTP requests
2	Number of GET requests
3	Number of POST requests
4	Average length of the URLs
5	Average number of parameters in the request
6	Average amount of data sent by POST requests
7	Average response length.

Table 4: Feature extraction from HTTP trace

4.5.1.1 Machine learning algorithms

We use most popular machine learning algorithms that used in lectures for problems of malware family classifications. We applied Random forest, Decision tree, Nearest Neighbors and Support vector machine as classifiers. Obtained results are show in Chapter 5.

4.5.2 Classification based on structural similarity

We perform approach presented in [37]. The objective of this approach is to find families of malware that interact with the server in a similar way. Learn a network behavior model for each family of malware, and then use such models to detect the presence of compromised machines in a monitored network. To this aim, we first perform behavioral classification of malware samples by finding structural similarities between the sequences of HTTP requests generated as a consequence of infection.

We consider the structural similarity among sequences of HTTP requests (as opposed to the statistical similarity used in Section 4.5.1). Our approach is based on the observation that two different malware samples that rely on the same server application will query URLs structured in a similar way and in a similar sequence. In order to capture these similarities, we first define a measure of distance between two HTTP requests r_k and r_h generated by two different malware samples. Consider Figure 18, where m, p, n and v, represent method, page, parameter name, parameter value that are different parts of an HTTP request:

- $\boldsymbol{m}$ Represents the request method (e.g., GET, POST). Is defined a distance function $\boldsymbol{d_m}\ (\boldsymbol{r_k}\,, \boldsymbol{r_h})$ that is equal to 0 if the requests r_k and $\boldsymbol{r_h}$ both use the same method (e.g., both are GET requests), otherwise it is equal to 1.

- $\boldsymbol{p}$ Stands for page, namely the first part of the URL that includes the path and page name, but does not include the parameters. Is defined $d_p(r_k\,, r_h)$to be equal to the Levenshtein distance[10] between the strings related to the path and pages that appear in the two requests $r_k$and$\boldsymbol{r_h}$. Levenshtein distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.
- $\boldsymbol{n}$ Represents the set of parameter names $(eg.n = \{id\,, version\,, cc\,\})$ in the example in Figure 18. Is defined $d_n(r_k\,, r_h)$ as the Jaccard distance[11] between the sets of parameters names in the two requests. The Jaccard distance, is a statistic used for comparing the similarity and diversity of sample sets. The Jaccard distance measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets.

- $\boldsymbol{v}$ Is the set of parameter values. Is defined $\boldsymbol{d_v\,(r_k\,, r_h)}$ to be equal to the Levenshtein distance between strings obtained by concatenating the parameter values (e.g., 0011. in Figure 18).

Is defined the overall distance between two HTTP requests as:

$$\boldsymbol{d_r\,(r_k\,, r_h) = w_m\; d_m\,(r_k\,, r_h)\; + w_p\; d_p\,(r_k\,, r_h)\; +\; w_n\; d_n\,(r_k\,, r_h)\; + w_v\; d_v\,(r_k\,, r_h)}$$

Where the factors $w_x\,, x \in \{m, p, n, v\}$ are predefined weights (the actual value assigned to the weightsw_x) that give more importance to the distance between the request methods and pages, for example, and less weight to the distance between parameter values. Is defined distance between two malware samples as the average minimum distance between sequences of HTTP requests from the two samples. We must select one classification algorithm to be on base of distance measure.

[10] https://en.wikipedia.org/wiki/Levenshtein_distance

[11] https://en.wikipedia.org/wiki/Jaccard_index

One common classification scheme based on the use of distance measures is that of the K-NN (see Section 2.5.5). We use this algorithm with k=10 as parameter of distance. The obtained result with this approach is discussed with chapter 5.

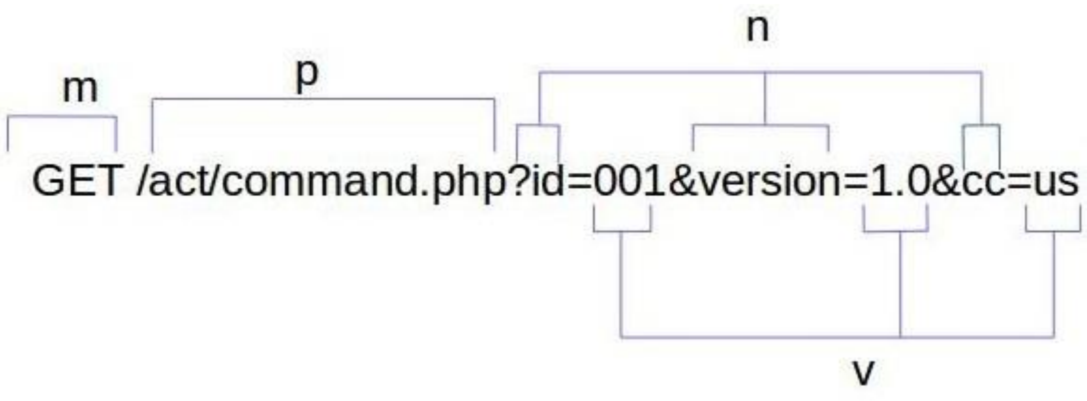


Figure 18: structure of a HTTP request used in structural similarity method

(m= Method; p= Page; n= Parameter Names; v= Parameter Values)

Chapter 5

Experiment and Results

In this chapter, we discuss the results of the experiments, evaluate the performance of the classification models, and demonstrate our findings. The first section explains the terminology of the metrics that we used in measuring the performance of two proposed methods for binary classification (malware from benign apps). The second section shows the obtained results for two proposed methods for malware family classification with our dataset.

5.1 Experiment environment

Due to the large dataset and its high-dimensional features space that require powerful computational resources for applying variety of machine learning algorithms, we did carry out our experiments with characteristics that are shown in the table below.

CPU	Intel CORE i5-520M Processor 2.40 GHZ
Memory	4 GB
OS	Linux Ubuntu 16.4

Table 5: Experiment environment

5.2 Accuracy of malware detection

In this section we demonstrate obtained results from malware detection on base of two proposed methods we explain first method in Section 4.3.1 and second method in Section 4.3.2. Figure 19 compares between obtained results first and second method for malware detection.

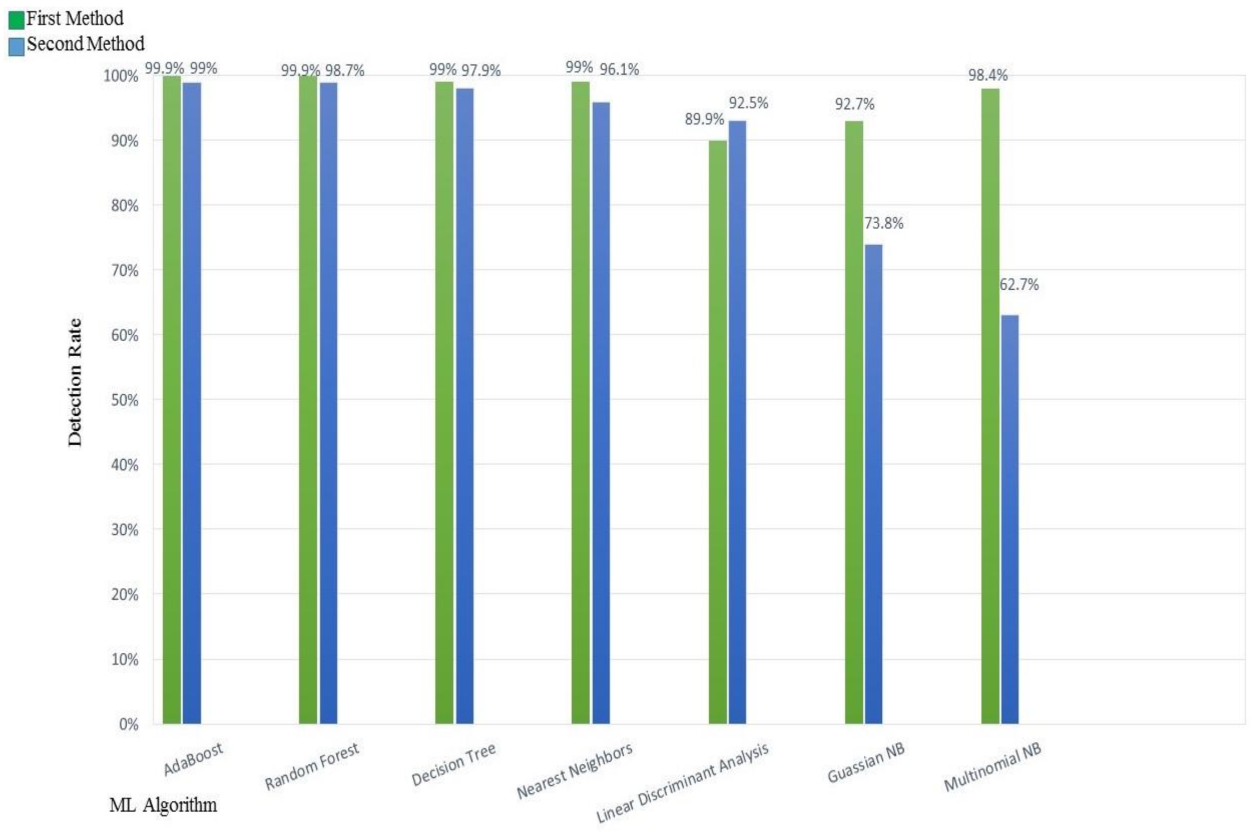


Figure 19: Compare accuracy of first and second method for malware detection

It is notable that looking Figure 19 we can claim that generally first proposed method we can detect malware better than second method.

5.3 Evaluation measures for malware detection

To evaluate the detection performance successfully, it is necessary to identify appropriate performance metrics. The following measures were derived from the confusion matrix to calculate and be applied to classifier evaluation. A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. In fact the confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives you insight not only into the errors being made by your classifier but more importantly the types of errors that are being made. It is this breakdown that overcomes the limitation of using classification accuracy alone.

On the main diagonal, you can view true negative (benign applications correctly recognized) and true positives (malware properly classified as such malware). In the other two cells are located false positives (benign app mistaken for malware) and false negatives (malware incorrectly classified as benign applications).

5.3.1 Evaluation

In order to evaluate the performance of the classification models, we used the metrics as the following:

- Accuracy: The proportion of the total number of the apps that are correctly classified whether as benign or malicious. $Accuracy = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}$
- Precision: The proportion of the actual malicious apps are correctly classified to the total of all apps that are classified as malicious. $Precision = \frac{t_p}{t_p + f_p}$

- Recall: The proportion of the malicious apps that are classified correctly to the total number of the malicious that are classified correctly as malicious or incorrectly as benign. $Recall = \frac{t_p}{t_p + f_n}$

- F-Measure: The harmonic mean of precision and recall. This value tells how much the model is discriminative $F - Measure = 2 . \frac{Precision .Recall}{Precision + Rcall}$

- True positive rate = $\frac{t_p}{t_p+f_n}$

- False positive rate = $\frac{f_p}{t_n+f_p}$

- True negative rate = $\frac{t_n}{t_n + f_p}$

- False negative rate =$\frac{f_n}{f_n + t_p}$

The next sections show the average performance of 100 iterations of testing the learned classifiers. In each round, the datasets were randomly shuffled using Bootstrapped method [40]: 90% for training and 10% for testing the classifier.

5.3.2 Evaluation Metric for first method

Metric	AdaBoost	Random Forest	Decision Tree	Nearest neighbor	LDA	Gaussian NB	Multinomial GN
Accuracy	**0.9999**	**0.9999**	0.9912	0.9869	0.9431	0.8072	0.9959
Precision	**0.9999**	**0.9999**	0.9982	0.9902	0.9036	**0.9999**	**0.9999**
F-Measure	**0.9999**	**0.9999**	0.9946	0.9885	0.9457	0.7612	0.9855
Recall	**0.9999**	**0.9999**	0.9841	0.9835	0.9920	0.6146	0.9920
FPR	0.00005	0.000035	0.0017	0.0097	0.1058	0.000049	**0.00004**
TPR	**0.9999**	**0.9999**	0.9841	0.9835	0.9920	0.6146	0.9920
FNR	**0.0001**	**0.0001**	0.0159	0.0165	0.80	0.3854	0.1830
TNR	0.99995	**0.999965**	0.9983	0.9903	0.8942	0.999951	0.99996

Table 6: Evaluation metric of first method for malware detection

5.3.3 Evaluation Metric for second method

Metric	AdaBoost	Random Forest	Decision Tree	Nearest neighbor	LDA	Gaussian NB	Multinomial GN
Accuracy	**0.9927**	0.9887	0.9820	0.9602	0.9276	0.7399	0.6164
Precision	**0.9916**	0.9838	0.9850	0.9596	0.9459	0.9809	0.4735
F-Measure	**0.9925**	0.9888	0.9798	0.9600	0.9288	0.7927	0.5503
Recall	0.9937	**0.9939**	0.9784	0.9605	0.9124	0.6595	0.6572
FPR	**0.0081**	0.0163	0.0144	0.0400	0.0559	0.0128	0.4062
TPR	0.9937	**0.9939**	0.9784	0.9605	0.9124	0.6595	0.6572
FNR	0.0062	**0.0060**	0.0215	0.0394	0.0875	0.3404	0.3427
TNR	**0.9918**	0.9836	0.9855	0.9599	0.9440	0.9871	0.5937

Table 7: Evaluation metric of second method for malware detection

Viewing the results, it is seen that the highest concentration of high values lie between classifier Viewing obtained results in both of them methods we can say first proposed method is better method; both aspect accuracy and evolution of rate of error.

5.4 Experiment results for family classification

This section is divided in two, where the first part discusses the results from statistical similarity method of family classification and second part discusses the results from structural similarity method of family classification.

5.4.1 Accuracy of statistical similarity

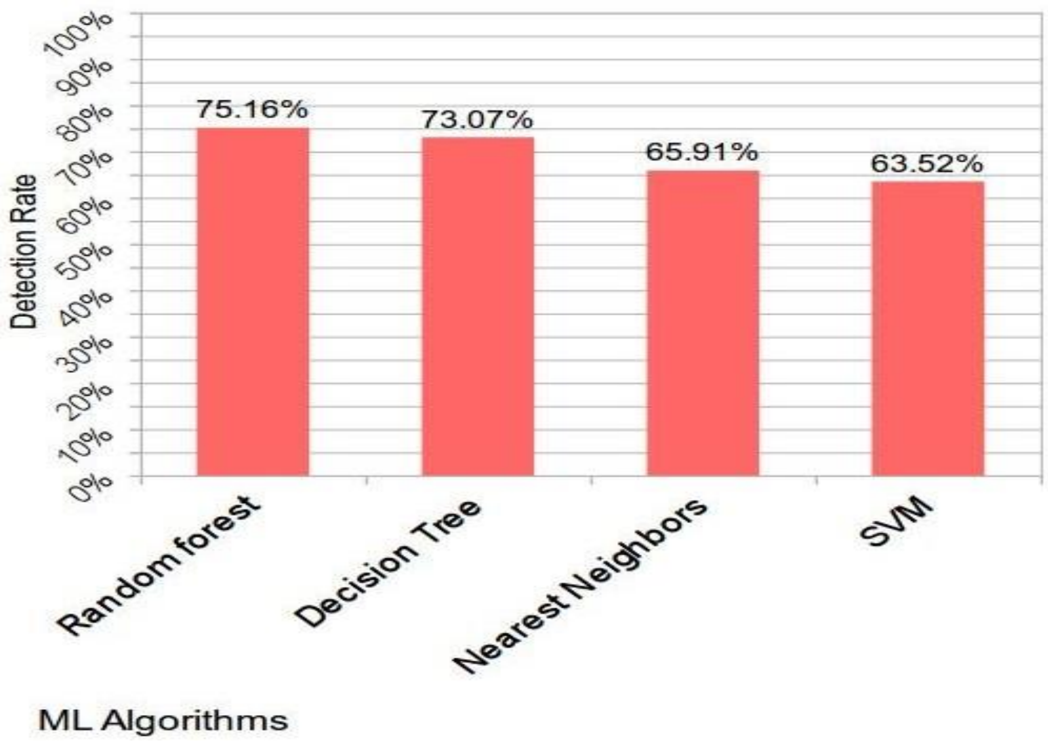


Figure 20: Detection rate of family classification

5.4.2 Calculate of confusion matrix for statistical similarity

The confusion matrix is a standard output representation of classification problems. It plots the true class of instances along of y-axis in a classification problem against the predicted class along of x-axis. It gives the predicted distribution of the test instances into each of the trained classes. For a n-class classification task, it is a n×n matrix, with entries in row I summing up to the number of test instances in class i. Ideally, with 100% accuracy, the n×n matrix only has diagonal entries corresponding to all test samples being correctly predicted to their true class. In reality, the confusion matrix is often smudged with small values being distributed all over. In our experience, the confusion matrix gives much more information about the nature of the classification problem involved.

Figure 21 shows a sample of confusion matrix with 23 classes in malware family categorization problem. The dataset contains 814 malware samples from 23 families. We calculate confusion matrix using Bootstrapped techniques [40] with select experimental parameter, number of Boost equal to 100 and using 10 percentage of my dataset as test set.

5.4.2.1 Observations from confusion matrix

A large number of test instances of a particular class may be misclassified into another class or a set of classes. From domain knowledge we often know that some classes are similar to each other. In the setting of the classification problem it is not unreasonable to expect that these similar classes will ‘confuse’ amongst each other. That means that if A and B are two classes similar to each other, then test instances from A, when misclassified, get predicted quite often as B and vice-versa. A and B are thus said to be confusing classes.

For example Figure 21 is a confusion matrix for random forest classifier, For the 23 classes samples, from domain knowledge we know that the malware samples in the class AVPass 82.3

percentage true predicted and are apt to be confused with samples in the class BaseBridge(0.9**)**, DroidKungFu (11.5) and Extension(3.5)**,** FakeAngry (0.9)**,** SkullKey (0.9). Similarity,samples in Extension class are more likely to be similar to samples in Fake Angry class than any of the other classes.

As seen in figures 21, 22, 23, 24 we calculated confusion matrix for other classifier algorithms. We can interpreted them, like Random forest algorithm.

5.4.2.2 Random Forest

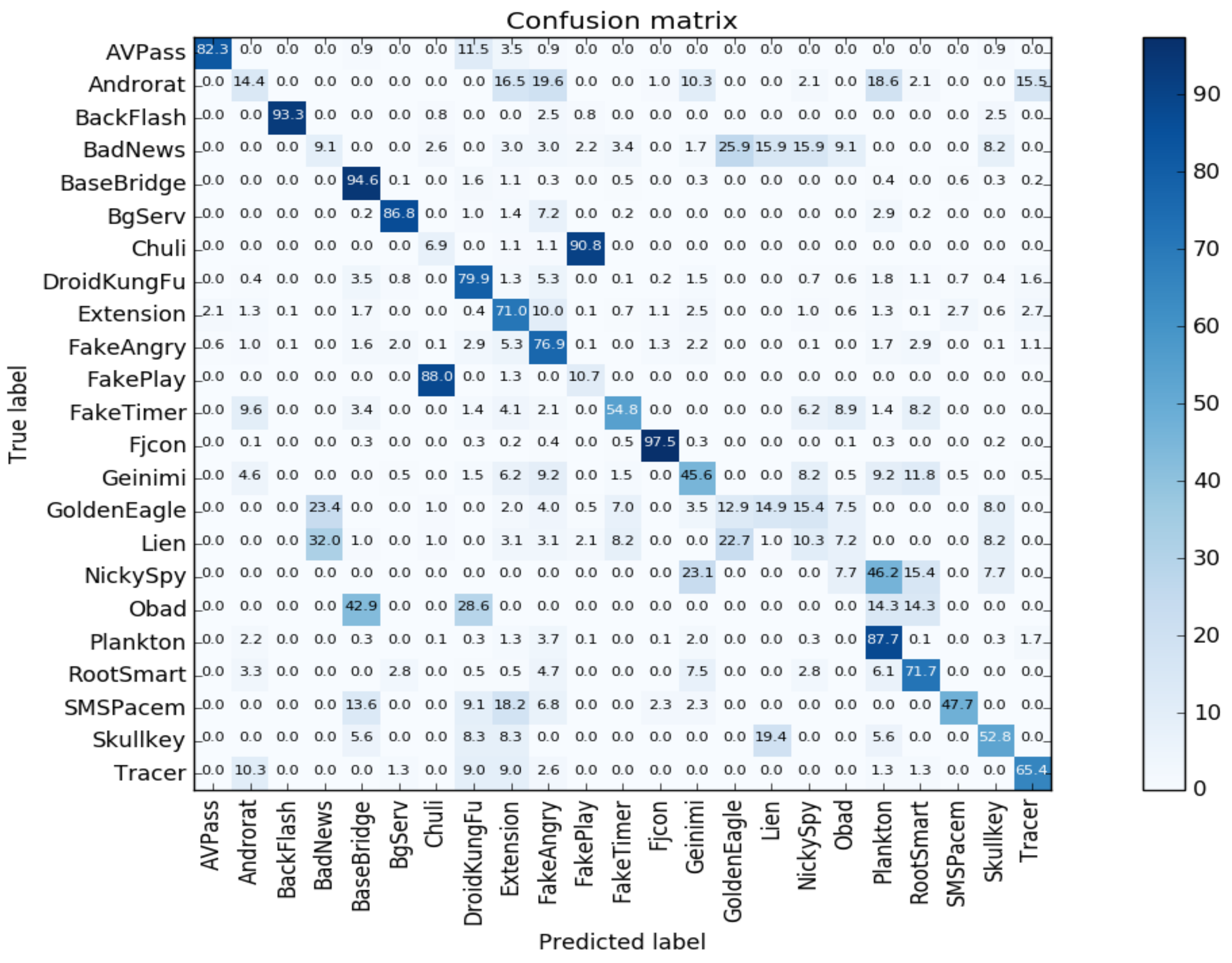
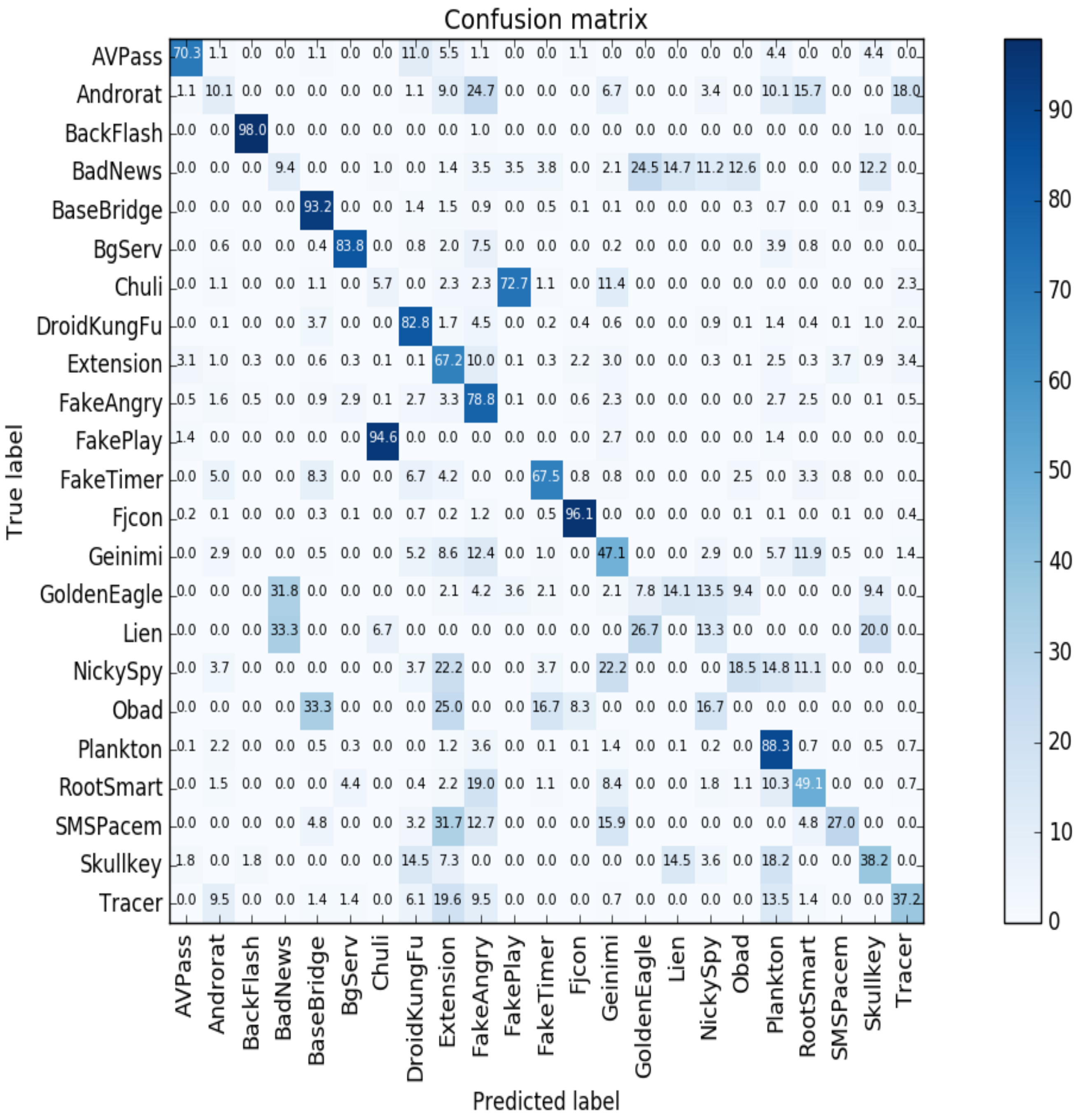


Figure 21: Confusion matrix of Random Forest for family classification

5.4.2.3 Decision Tree



True label \ Predicted label	AVPass	Androrat	BackFlash	BadNews	BaseBridge	BgServ	Chuli	DroidKungFu	Extension	FakeAngry	FakePlay	FakeTimer	Fjcon	Geinimi	GoldenEagle	Lien	NickySpy	Obad	Plankton	RootSmart	SMSPacem	Skullkey	Tracer
AVPass	70.3	1.1	0.0	0.0	1.1	0.0	0.0	11.0	5.5	1.1	0.0	0.0	1.1	0.0	0.0	0.0	0.0	0.0	4.4	0.0	0.0	4.4	0.0
Androrat	1.1	10.1	0.0	0.0	0.0	0.0	0.0	1.1	9.0	24.7	0.0	0.0	0.0	6.7	0.0	0.0	3.4	0.0	10.1	15.7	0.0	0.0	18.0
BackFlash	0.0	0.0	98.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
BadNews	0.0	0.0	0.0	9.4	0.0	0.0	1.0	0.0	1.4	3.5	3.5	3.8	0.0	2.1	24.5	14.7	11.2	12.6	0.0	0.0	0.0	12.2	0.0
BaseBridge	0.0	0.0	0.0	0.0	93.2	0.0	0.0	1.4	1.5	0.9	0.0	0.5	0.1	0.1	0.0	0.0	0.0	0.3	0.7	0.0	0.1	0.9	0.3
BgServ	0.0	0.6	0.0	0.0	0.4	83.8	0.0	0.8	2.0	7.5	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	3.9	0.8	0.0	0.0	0.0
Chuli	0.0	1.1	0.0	0.0	1.1	0.0	5.7	0.0	2.3	2.3	72.7	1.1	0.0	11.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.3
DroidKungFu	0.0	0.1	0.0	0.0	3.7	0.0	0.0	82.8	1.7	4.5	0.0	0.2	0.4	0.6	0.0	0.0	0.9	0.1	1.4	0.4	0.1	1.0	2.0
Extension	3.1	1.0	0.3	0.0	0.6	0.3	0.1	0.1	67.2	10.0	0.1	0.3	2.2	3.0	0.0	0.0	0.3	0.1	2.5	0.3	3.7	0.9	3.4
FakeAngry	0.5	1.6	0.5	0.0	0.9	2.9	0.1	2.7	3.3	78.8	0.1	0.0	0.6	2.3	0.0	0.0	0.0	0.0	2.7	2.5	0.0	0.1	0.5
FakePlay	1.4	0.0	0.0	0.0	0.0	0.0	94.6	0.0	0.0	0.0	0.0	0.0	0.0	2.7	0.0	0.0	0.0	0.0	1.4	0.0	0.0	0.0	0.0
FakeTimer	0.0	5.0	0.0	0.0	8.3	0.0	0.0	6.7	4.2	0.0	0.0	67.5	0.8	0.8	0.0	0.0	0.0	2.5	0.0	3.3	0.8	0.0	0.0
Fjcon	0.2	0.1	0.0	0.0	0.3	0.1	0.0	0.7	0.2	1.2	0.0	0.5	96.1	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.1	0.0	0.4
Geinimi	0.0	2.9	0.0	0.0	0.5	0.0	0.0	5.2	8.6	12.4	0.0	1.0	0.0	47.1	0.0	0.0	2.9	0.0	5.7	11.9	0.5	0.0	1.4
GoldenEagle	0.0	0.0	0.0	31.8	0.0	0.0	0.0	0.0	2.1	4.2	3.6	2.1	0.0	2.1	7.8	14.1	13.5	9.4	0.0	0.0	0.0	9.4	0.0
Lien	0.0	0.0	0.0	33.3	0.0	0.0	6.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	26.7	0.0	13.3	0.0	0.0	0.0	0.0	20.0	0.0
NickySpy	0.0	3.7	0.0	0.0	0.0	0.0	0.0	3.7	22.2	0.0	0.0	3.7	0.0	22.2	0.0	0.0	0.0	18.5	14.8	11.1	0.0	0.0	0.0
Obad	0.0	0.0	0.0	0.0	33.3	0.0	0.0	0.0	25.0	0.0	0.0	16.7	8.3	0.0	0.0	0.0	16.7	0.0	0.0	0.0	0.0	0.0	0.0
Plankton	0.1	2.2	0.0	0.0	0.5	0.3	0.0	0.0	1.2	3.6	0.0	0.1	0.1	1.4	0.0	0.1	0.2	0.0	88.3	0.7	0.0	0.5	0.7
RootSmart	0.0	1.5	0.0	0.0	0.0	4.4	0.0	0.4	2.2	19.0	0.0	1.1	0.0	8.4	0.0	0.0	1.8	1.1	10.3	49.1	0.0	0.0	0.7
SMSPacem	0.0	0.0	0.0	0.0	4.8	0.0	0.0	3.2	31.7	12.7	0.0	0.0	0.0	15.9	0.0	0.0	0.0	0.0	0.0	4.8	27.0	0.0	0.0
Skullkey	1.8	0.0	1.8	0.0	0.0	0.0	0.0	14.5	7.3	0.0	0.0	0.0	0.0	0.0	0.0	14.5	3.6	0.0	18.2	0.0	0.0	38.2	0.0
Tracer	0.0	9.5	0.0	0.0	1.4	1.4	0.0	6.1	19.6	9.5	0.0	0.0	0.0	0.7	0.0	0.0	0.0	0.0	13.5	1.4	0.0	0.0	37.2

Figure 22: Confusion matrix of Decision Tree for family classification

5.4.2.4 Nearest Neighbors

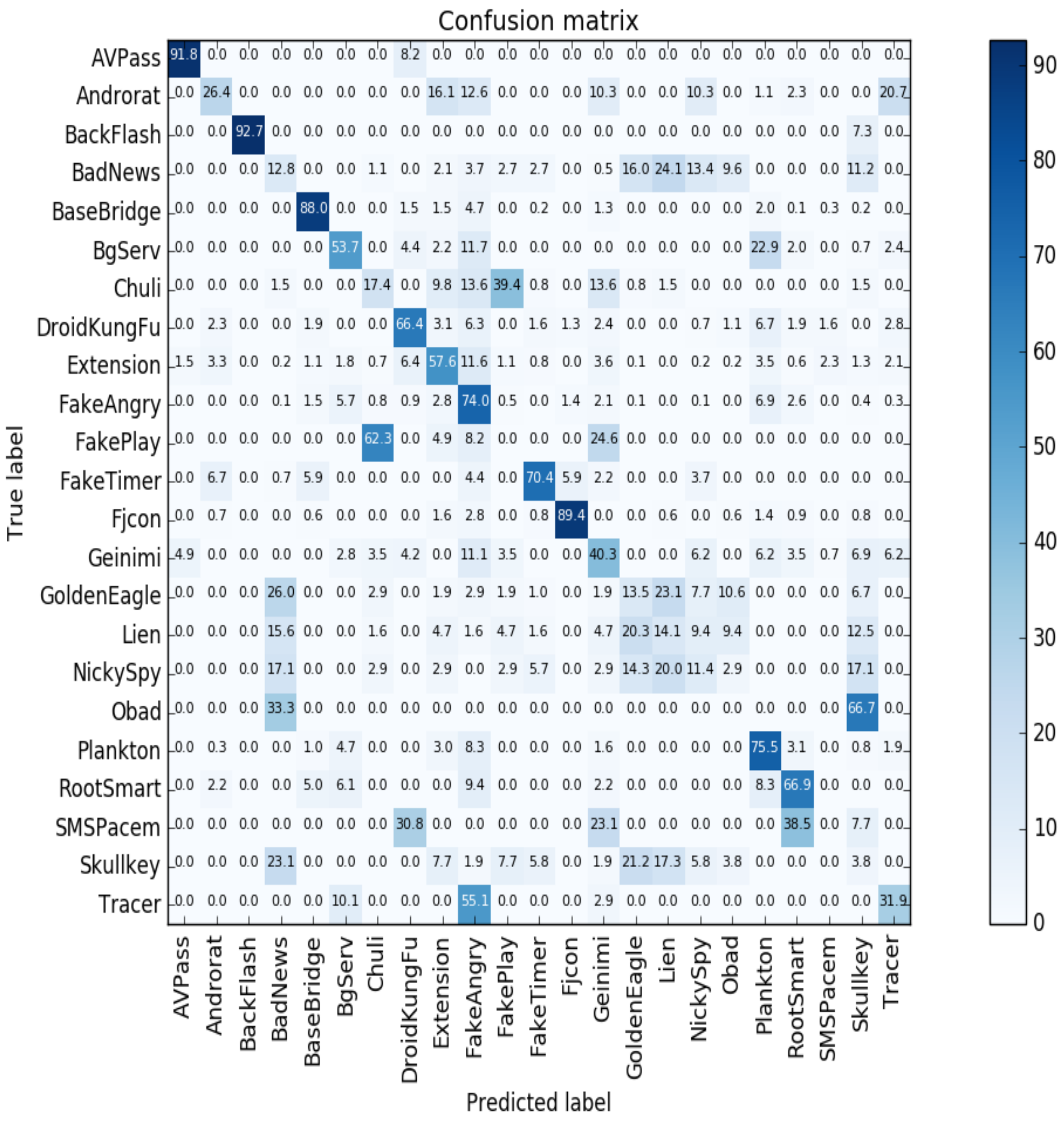
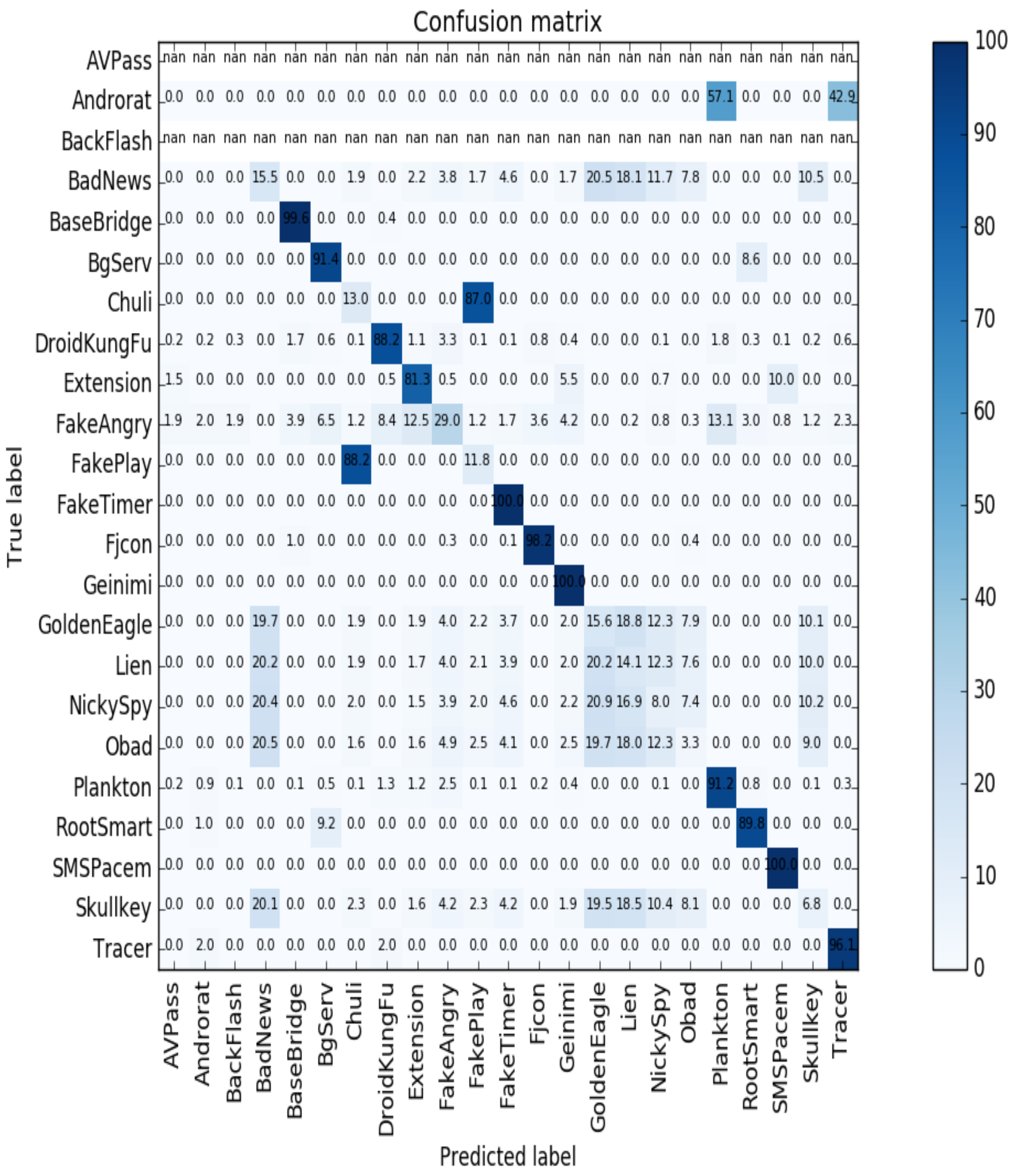


Figure 23: Confusion matrix of Nearest Neighbors for family classification

5.4.2.5 Support Vector Machine



	AVPass	Androrat	BackFlash	BadNews	BaseBridge	BgServ	Chuli	DroidKungFu	Extension	FakeAngry	FakePlay	FakeTimer	Fjcon	Geinimi	GoldenEagle	Lien	NickySpy	Obad	Plankton	RootSmart	SMSPacem	Skullkey	Tracer
AVPass	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
Androrat	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	57.1	0.0	0.0	0.0	42.9
BackFlash	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
BadNews	0.0	0.0	0.0	15.5	0.0	0.0	1.9	0.0	2.2	3.8	1.7	4.6	0.0	1.7	20.5	18.1	11.7	7.8	0.0	0.0	0.0	10.5	0.0
BaseBridge	0.0	0.0	0.0	0.0	99.6	0.0	0.0	0.4	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
BgServ	0.0	0.0	0.0	0.0	0.0	91.4	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	8.6	0.0	0.0	0.0
Chuli	0.0	0.0	0.0	0.0	0.0	0.0	13.0	0.0	0.0	0.0	87.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DroidKungFu	0.2	0.2	0.3	0.0	1.7	0.6	0.1	88.2	1.1	3.3	0.1	0.1	0.8	0.4	0.0	0.0	0.1	0.0	1.8	0.3	0.1	0.2	0.6
Extension	1.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	81.3	0.5	0.0	0.0	0.0	5.5	0.0	0.0	0.7	0.0	0.0	0.0	10.0	0.0	0.0
FakeAngry	1.9	2.0	1.9	0.0	3.9	6.5	1.2	8.4	12.5	29.0	1.2	1.7	3.6	4.2	0.0	0.2	0.8	0.3	13.1	3.0	0.8	1.2	2.3
FakePlay	0.0	0.0	0.0	0.0	0.0	0.0	88.2	0.0	0.0	0.0	11.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
FakeTimer	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Fjcon	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.3	0.0	0.1	98.2	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0
Geinimi	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	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
GoldenEagle	0.0	0.0	0.0	19.7	0.0	0.0	1.9	0.0	1.9	4.0	2.2	3.7	0.0	2.0	15.6	18.8	12.3	7.9	0.0	0.0	0.0	10.1	0.0
Lien	0.0	0.0	0.0	20.2	0.0	0.0	1.9	0.0	1.7	4.0	2.1	3.9	0.0	2.0	20.2	14.1	12.3	7.6	0.0	0.0	0.0	10.0	0.0
NickySpy	0.0	0.0	0.0	20.4	0.0	0.0	2.0	0.0	1.5	3.9	2.0	4.6	0.0	2.2	20.9	16.9	8.0	7.4	0.0	0.0	0.0	10.2	0.0
Obad	0.0	0.0	0.0	20.5	0.0	0.0	1.6	0.0	1.6	4.9	2.5	4.1	0.0	2.5	19.7	18.0	12.3	3.3	0.0	0.0	0.0	9.0	0.0
Plankton	0.2	0.9	0.1	0.0	0.1	0.5	0.1	1.3	1.2	2.5	0.1	0.1	0.2	0.4	0.0	0.0	0.1	0.0	91.2	0.8	0.0	0.1	0.3
RootSmart	0.0	1.0	0.0	0.0	0.0	9.2	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	89.8	0.0	0.0	0.0
SMSPacem	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	100.0	0.0	0.0
Skullkey	0.0	0.0	0.0	20.1	0.0	0.0	2.3	0.0	1.6	4.2	2.3	4.2	0.0	1.9	19.5	18.5	10.4	8.1	0.0	0.0	0.0	6.8	0.0
Tracer	0.0	2.0	0.0	0.0	0.0	0.0	0.0	2.0	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	96.1

Figure 24: Confusion matrix of SVM for family classification

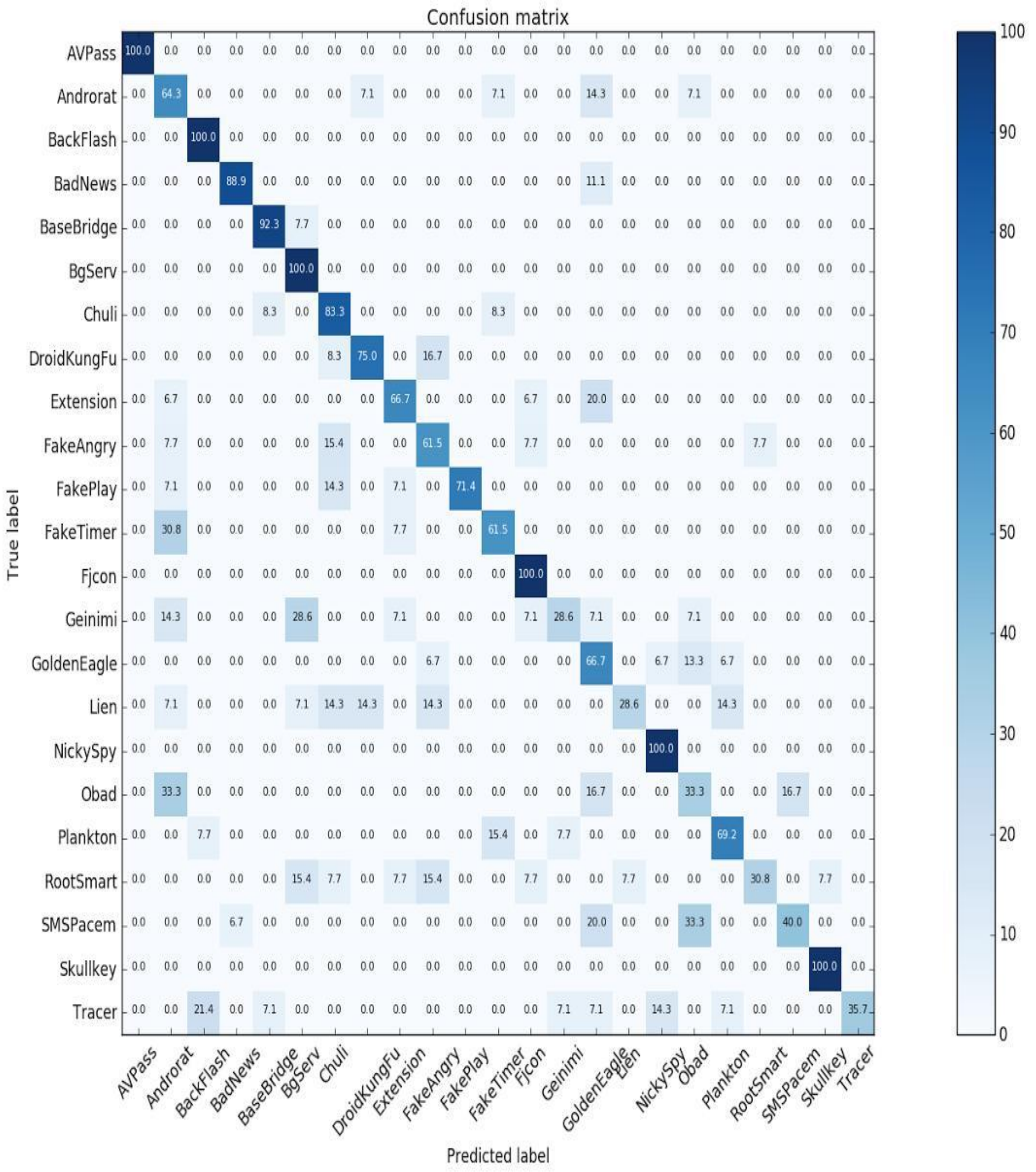
An important point we want mentioned here is that, by analyzing the all confusion matrix (Figures 21, 22, 23, 24) we noticed that ,poor results are related to classes that lack of data; For example classes AVPass, BadNews, Nickispy and Obad each one contains 3 samples so all classifiers tested in our work give poor results; Vice versa classes with appropriate number of samples give acceptable results like BaseBridge (112 samples), BgServ (45 samples), DroidKungFu (86 samples), Fjcon (106 samples), Plankton (119 samples) . So we can claim, the problem of poor results in certain classes is not for our proposed approach or classifier algorithms; this problem related to dataset and samples of classes that are few in certain classes.

5.4.3 Results obtained from structural similarity

More generally, the k nearest neighbors are computed, and the new example is assigned the class that is most frequent among these k neighbors (this will be abbreviated as KNN). Since base of this approach is calculation of Euclidean distance between http traces we were mandatory to use algorithm for classification that is on base of distance. The K-NN [7] is one of the oldest and simplest methods for pattern classification we explain KNN in Section 2.5.5. We select parameter k=10 and we perform this classifier on our dataset and we obtained **%72.61** as result.

5.4.3.1 Calculate of confusion matrix

Like section 5.5 we calculate confusion matrix for KNN algorithm .obtained results has been seen in figure 25.



	AVPass	Androrat	BackFlash	BadNews	BaseBridge	BgServ	Chuli	DroidKungFu	Extension	FakeAngry	FakePlay	FakeTimer	Fjcon	Geinimi	GoldenEagle	Lien	NickySpy	Obad	Plankton	RootSmart	SMSPacem	Skullkey	Tracer
AVPass	100.0	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
Androrat	0.0	64.3	0.0	0.0	0.0	0.0	0.0	7.1	0.0	0.0	0.0	7.1	0.0	0.0	14.3	0.0	0.0	7.1	0.0	0.0	0.0	0.0	0.0
BackFlash	0.0	0.0	100.0	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
BadNews	0.0	0.0	0.0	88.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	11.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
BaseBridge	0.0	0.0	0.0	0.0	92.3	7.7	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
BgServ	0.0	0.0	0.0	0.0	0.0	100.0	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
Chuli	0.0	0.0	0.0	0.0	8.3	0.0	83.3	0.0	0.0	0.0	0.0	8.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DroidKungFu	0.0	0.0	0.0	0.0	0.0	0.0	8.3	75.0	0.0	16.7	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
Extension	0.0	6.7	0.0	0.0	0.0	0.0	0.0	0.0	66.7	0.0	0.0	0.0	6.7	0.0	20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
FakeAngry	0.0	7.7	0.0	0.0	0.0	0.0	15.4	0.0	0.0	61.5	0.0	0.0	7.7	0.0	0.0	0.0	0.0	0.0	0.0	7.7	0.0	0.0	0.0
FakePlay	0.0	7.1	0.0	0.0	0.0	0.0	14.3	0.0	7.1	0.0	71.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
FakeTimer	0.0	30.8	0.0	0.0	0.0	0.0	0.0	0.0	7.7	0.0	0.0	61.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Fjcon	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Geinimi	0.0	14.3	0.0	0.0	0.0	28.6	0.0	0.0	7.1	0.0	0.0	0.0	7.1	28.6	7.1	0.0	0.0	7.1	0.0	0.0	0.0	0.0	0.0
GoldenEagle	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	6.7	0.0	0.0	0.0	0.0	66.7	0.0	6.7	13.3	6.7	0.0	0.0	0.0	0.0
Lien	0.0	7.1	0.0	0.0	0.0	7.1	14.3	14.3	0.0	14.3	0.0	0.0	0.0	0.0	0.0	28.6	0.0	0.0	14.3	0.0	0.0	0.0	0.0
NickySpy	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	100.0	0.0	0.0	0.0	0.0	0.0	0.0
Obad	0.0	33.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	16.7	0.0	0.0	33.3	0.0	0.0	16.7	0.0	0.0
Plankton	0.0	0.0	7.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	15.4	0.0	7.7	0.0	0.0	0.0	0.0	69.2	0.0	0.0	0.0	0.0
RootSmart	0.0	0.0	0.0	0.0	0.0	15.4	7.7	0.0	7.7	15.4	0.0	0.0	7.7	0.0	0.0	7.7	0.0	0.0	0.0	30.8	0.0	7.7	0.0
SMSPacem	0.0	0.0	0.0	6.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	20.0	0.0	0.0	33.3	0.0	0.0	40.0	0.0	0.0
Skullkey	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	100.0	0.0
Tracer	0.0	0.0	21.4	0.0	7.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	7.1	7.1	0.0	14.3	0.0	7.1	0.0	0.0	0.0	35.7

Figure 25: Confusion matrix of KNN for family classification

Chapter 6

Conclusion and future work

In this thesis we have two important aim first we wanted to detection botnets malware from benign applications and in then we put one step further and we classified our malwares dataset on a base of family.

In fact this thesis is divided in two part**:** our goals in first part is detection of malware from benign application that leads to binary classification. For this aim we showed two separated method for malware detection.

In the second part of thesis we performed an analysis of Android botnets that employ HTTP traffic for their communications. By classification the generated network traffic of different Android malwares usage machine learning algorithms. We considered two methods to detection of botnets malwares on base of their families. In the first proposed method we presented a network-level behavioral malware classification system that focuses on HTTP-based malware and classification malware samples based on a notion of statistical similarity between the malicious HTTP traffic they generate. In the second method we showed that the samples belonging to the same malware family have structural similarity between HTTP malicious traffic traces.

The malware sample employed in this thesis is one of the newest in the research community [40]. However, with the advent of novel malware each month, it is imperative to collect malware samples continuously and to analyze and improve security systems. However the unavailability of a larger Android malware dataset remains a great problem in evaluating dynamic approaches. With a proper dataset shared among researchers, a system that learns a new malware and share that knowledge to all the mobile devices, so that they can protect themselves from future attacks, could be developed.

6.1 Future work

However, we conclude that this approach has opportunities which could be more explored. For future work, the combined feature set can be evaluated using different machine learning techniques. We can also evaluate the effectiveness of the above approach using a larger dataset. We can implement hybrid solution which will be a generic antimalware that will provide better security for Android devices by firstly statically analyzing the Android applications and then it will perform dynamic analysis on traffic of network. In this way we can take advantage of both dynamic and static analysis.

Bibliography:

[1] https://securelist.com/analysis/quarterly-malware-reports/76513/it-threat-evolution-q3-2016 - statistics/

[2] https://securelist.com/analysis/quarterly-malware-reports/71610/it-threat-evolution-q2-2015/

[3] Safavian, S. Rasoul, and David Landgrebe. "A survey of decision tree classifier methodology. " IEEE transactions on systems, man, and cybernetics 21.3 (1991): 660-674.

[4] L. Breiman. Random Forests. Machine learning, 45(1):5–32, 2001. (Cited on pages 2, 69, 71, 72, 74, 81, 88, 103, 115, 124, 125, and 138.)

[5] Opitz, David, and Richard Maclin. "Popular ensemble methods: An empirical study." Journal of Artificial Intelligence Research 11 (1999): 169-198.

[6] S. Raschka, "Naive Bayes and Text Classification I - Introduction and Theory," 2014.

[7] C. M. Bishop, Pattern Recognition and Machine Learning.Springer-Verlag New York, Inc. 2006.

[8] A. Mayr, H. Binder, O. Gefeller, M. Schmid et al., "The evolution of boosting algorithms," Methods of information in medicine, vol. 53, no. 6, pp. 419–427, 2014.

[9] Schiller, C. A., Binkley, J., Harley, D., Evron, G., Bradley, T., Willems, C., & Cross, M. (2007). Botnet The Killer Web App. Rockland, MA: Syngress.

[10] Heron, S. (2007). Working the botnet: how dynamic DNS is revitalising the zombie army. Network Security, 2007(1), 9-11. doi:10.1016/s1353-4858(07)70005-3

[11] Dietrich, C. J., Rossow, C., & Pohlmann, N. (2013). CoCoSpot: Clustering and recognizing botnet command and control channels using traffic analysis.

[12] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," Proc. 1st ACM Work.Secur. Priv. smartphones Mob. devices - SPSM "11, p. 15, 2011.

[13] "strace downloadSourceForge.net." [Online]. Available:http://sourceforge.net/projects/ strac e/ . [Accessed: 22-Dec-2015].

[14] A. Shabtai, U. Kanonov , Y. Elovici, C. Glezer, and Y. Weiss,"„Andromaly‟: a behavioral malware detection framework for android devices," J. Intell. Inf. Syst., vol. 38, no. 1, pp. 161–190, 2012.

[15] M. Zhao, F. Ge, T. Zhang, and Z. Yuan, "Anti MalDroid: An efficient SVM-based malware detection framework for android," Commun. Comput. Inf. Sci., vol. 243 CCIS, pp. 158–166, 2011.

[16] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering android malware families by http traffic. In MALWARE, pages 128–135, 2015.

[17] Arora, Anshul, Shree Garg, and Sateesh K. Peddoju. "Malware detection using network traffic analysis in android based mobile devices." Next generation mobile apps, services and technologies (NGMAST), 2014 eighth international conference on. IEEE, 2014.

[18] L. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," Proc. 21st USENIX Secur. Symp., p. 29, 2012.

[19] F. Wu, H. Narang, and D. Clarke, "An Overview of Mobile Malware and Solutions," J. Comput. Commun., vol. 2, no. 2, pp. 8–17, 2014.

[20] T. Bläsing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," Proc. 5th IEEE Int. Conf. Malicious Unwanted Software, Malware 2010, pp. 55–62, 2010.

[21] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in NDSS, 2014.

[22] X. Liu and J. Liu, "A Two-layered Permission-based Android Malware Detection Scheme," in Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on.IEEE, 2014, pp.142–148.

[23] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission Usage to detect Malware in Android," in International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions.Springer, 2013, pp. 289–298.

[24] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection," Proc. 6th Int. Conf. Secur. Inf. Networks, pp. 152159, 2013.

[25] M. Zheng, M. Sun, and J. C. S. Lui, "DroidAnalytics : A Signature Based Analytic System to Collect , Extract , Analyze and Associate Android Malware," 2013.

[26] https://www.wireshark.org/docs/man-pages/tshark.html .

[27] X. J. Y. Zhou, “Android malware genome project,” http://www.malgenomeproject.org, 2012.

[28] M. Parkour,“Contagio mobile-mobile malware mini dump,”http:///contagiominidump .blogspot.it/, 2012.

[29] “Virustotal - free online virus, malware and url scanner,” https://www.virustotal.com/, 2014.

[30] “Anubis - free online malware analysis for unknown binaries (windows executable or android apk),” https://anubis.iseclab.org, 2014 .

[31] https://developer.android.com/studio/test/monkey.html .

[32] http://www.tcpdump.org/.

[33] A. Dries and U. Riickert, " Adaptive concept drift detection," Statistical Analy Data Mining, vol. 2, no. 5-6, pp. 311-327, 2009.

[34] Kohavi, Ron. "A study of cross-validation and bootstrap for accuracy estimation and model selection." Ijcai. Vol. 14. No. 2. 1995.

[35] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, “Networkpro- filer: Towards automatic fingerprinting of android apps,” in INFOCOM, 2013 Proceedings IEEE, April 2013, pp. 809–817.

[36] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Profiledroid: Multi-layer profiling of android applications,” in Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, ser. Mobicom ’12. New York, NY, USA: ACM, 2012, pp. 137–148. [Online]. Available: http://doi.acm.org/10.1145/2348543.2348563.

[37] Perdisci, Roberto, Wenke Lee, and Nick Feamster. "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces." *NSDI*. Vol. 10. 2010.

[38] Z. Aung and W. Zaw, “Permission-Based Android Malware Detection,” International Journal of Scientific and Technology Research, vol. 2, pp. 228–234,2013.

[39] A. Aiken, “Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis,” Fse 2014, pp. 576–587, 2014.

[40] Kohavi, Ron. "A study of cross-validation and bootstrap for accuracy estimation and model selection." *Ijcai*. Vol. 14. No. 2. 1995.