Behavioural Correlation for Malicious Bot Detection

by Yousof Ali Abdulla Al-Hammadi,

Thesis submitted to The University of Nottingham
for the degree of Doctor of Philosophy, April 2010

This thesis is dedicated to my family specially my wife, my mum and my dad for their love and support and encouragements. Thank You.

Acknowledgements

I would like to thank Emirates Telecommunications Corporation - ETISALAT and Khalifa University of Science, Technology and Research for their financial support and consistent encouragement to complete my PhD.

I would like express my sincere thankfulness to my supervisor Prof. Uwe Aickelin for his guidance, advise, support and encouragement during my PhD studies at the University of Nottingham. I would also like to thank Dr. Julie Greensmith for her help, comments and her guidance to the right direction which improve the quality of my research.

I also would like to thank my friends Jan Feyereisl, Feng Gu, Qi Chen and Gianni Tedecsco, Jamie Twycross and Robert Oates for their support, valuable assistance and time we spent together.

Contents

List of Tables

List of Figures

Abstract

Over the past few years, IRC bots, malicious programs which are remotely controlled by the attacker, have become a major threat to the Internet and its users. These bots can be used in different malicious ways such as to launch distributed denial of service (DDoS) attacks to shutdown other networks and services. New bots are implemented with extended features such as keystrokes logging, spamming, traffic sniffing, which cause serious disruption to targeted networks and users. In response to these threats, there is a growing demand for effective techniques to detect the presence of bots/botnets. Currently existing approaches detect botnets rather than individual bots. In our work we present a host-based behavioural approach for detecting bots/botnets based on correlating different activities generated by bots by monitoring function calls within a specified time window. Different correlation algorithms have been used in this work to achieve the required task. We start our work by detecting IRC bots' behaviours using a simple correlation algorithm. A more intelligent approach to understand correlating activities is also used as a major part of this work. Our intelligent algorithm is inspired by the immune system. Although the intelligent approach produces an anomaly value for the classification of processes, it generates false positive alarms if not enough data is provided. In order to solve this problem, we introduce a modified anomaly value which reduces the amount of false positives generated by the original anomaly value.

We also extend our work to detect peer to peer (P2P) bots which are the upcoming threat to Internet security due to the fact that P2P bots do not have a centralized point to shutdown or traceback, thus making the detection of P2P bots a real challenge. Our evaluation shows that correlating different activities generated by IRC/P2P bots within a specified time period achieves high detection accuracy. In addition, using an intelligent correlation algorithm not only states if an anomaly is present, but it also names the culprit responsible for the anomaly.

Chapter 1

Introduction

The Internet is persistently threatened by many types of attacks such as viruses, worms and trojan horses. These attacks have a negative impact on the Internet, and result in delays due to congestion, extensive waste of network bandwidth as well as corruption on users' computers and data. In addition, some of these attacks are used to control Internet hosts and then use these hosts to launch denial-of-service (DoS) attacks against other entities. If an attacker can gain access to network hosts, this can lead to enormous damage to the network such as disrupting e-commerce sites, news outlets, network infrastructure, routers and root name servers. An attacker can accelerate the process of gaining access to computers by using malicious software's such as viruses, worms or trojan horses. Viruses and worms exploit vulnerabilities in host software in order to automatically propagate between Internet hosts. These viruses and worms can carry an arbitrary malicious piece of software, called *bot*, for remote management. This can enable attackers to gain access to users' personal information like passwords, credit card numbers, confidential documents, address books, archived email or user activities. Moreover, these attacks could disturb, corrupt, or even change this information.

Recently, there have been several studies on how to detect botnets, group of bots, based on analysing network traffic or looking for well known bots signatures. These techniques have helped to understand different types of signatures that botnets use. We will discuss some of these techniques in Chapter 2, Section 2.5. However, the problem of detecting and reacting to an individual bot running on the system remains unsolved. While there are several techniques available for detecting the signatures of botnets using signature-based detection discussed in Section 2.5.3, these techniques are unable to detect new types of attacks carried out by the bots due to the lack of prior signatures. An alternative approach is to use anomaly detection, which tries to detect abnormal patterns of behaviour. An open problem is how to develop an anomaly detection technique that can efficiently and accurately detect the activity of botnets/bots.

Even though many techniques using anomaly detection have been proposed as discussed in Section 2.5.4, they focus on detecting botnets using network-based anomaly detection or they ignore detecting an individual bot running on the system by correlating different behaviours. In order to overcome these problems, we present different approaches to detect botnets/bots based on correlating different activities on the system.

1.1 Motivation

One of the most important issues in network security that the administrators have to handle carefully is the existence of bots/botnets. In this work, our main interest is to detect the existence of this kind of threat. This is due to many reasons which are listed below:

- Data regarding bots is rarely available. This is due to the fact that the botnet data can contain sensitive information. As a result, making this data publicly available can be dangerous.

- Little research has been conducted in order to detect botnets using anomaly detection for an individual bot running on the system. Previous research focuses on detecting botnets using non-productive resources, called honeypots. Furthermore recent research by others tries to detect the botnet by analysing network traffic looking for known botnet signatures. This is a classical way of detecting malicious patterns on the network. In addition, most of the existing research work focuses on detecting botnets rather than an individual bot.

- Botnets pose a severe threat to Internet security. For example, an army of several thousand bots can exhaust the bandwidth of a large number of systems or networks. An attacker can use the botnet to perform malicious activities [9]. These activities include performing DDoS which causes a loss of connectivity or services to legitimate users by consuming large amounts of the victim's bandwidth or overloading the computational resources of the victim's host. Another malicious activity include spreading massive *email spam*. Mcafee reports that the percentage of spam emails have increased from approximately 30% in the second quarter of 2006 to more than 80% in the same quarter of 2008 [104]. MessageLabs reports that botnets are responsible for distributing 87.9% of all spams and there is an increase of 2.9% from the second quarter of 2009 [105] and most of these spam emails come from the botnets. In addition, the attacker can use *phishing* e-mails to expand his botnets.

  A third malicious activity is *identity theft*. Identity theft is performed in two ways. The bot master can use packet sniffers to watch for the interesting cleartext data passing by the victim's machine, thus, retrieving sensitive information such as user name and password. Another way of having sensitive information is by implementing a *keylogger* which records all keystrokes typed, screen, or websites visited and send them to the attacker. The concept and the implementation of the keylogger are presented in Chapter 5.

  Another important activity used by attackers is to make revenue by *extorting* on-line business companies. In addition, the attacker can make revenue by renting his botnets to other malicious users. Furthermore, the attacker can use the bot to disable the Anti-Virus processes on the infected machines.

- During the past few years, there was an exponential growth of using botnets to perform large number of malicious activities ranging from Internet Relay Chat (IRC) bots and Peer-to-Peer (P2P) bots as shown in Figure 1.1. Mcafee [104] also reports that they have observed fourteen million new bots in the second quarter of the year 2009, in comparison to nearly twelve million new bots in the first quarter in the same year which is approximately an increase of more than

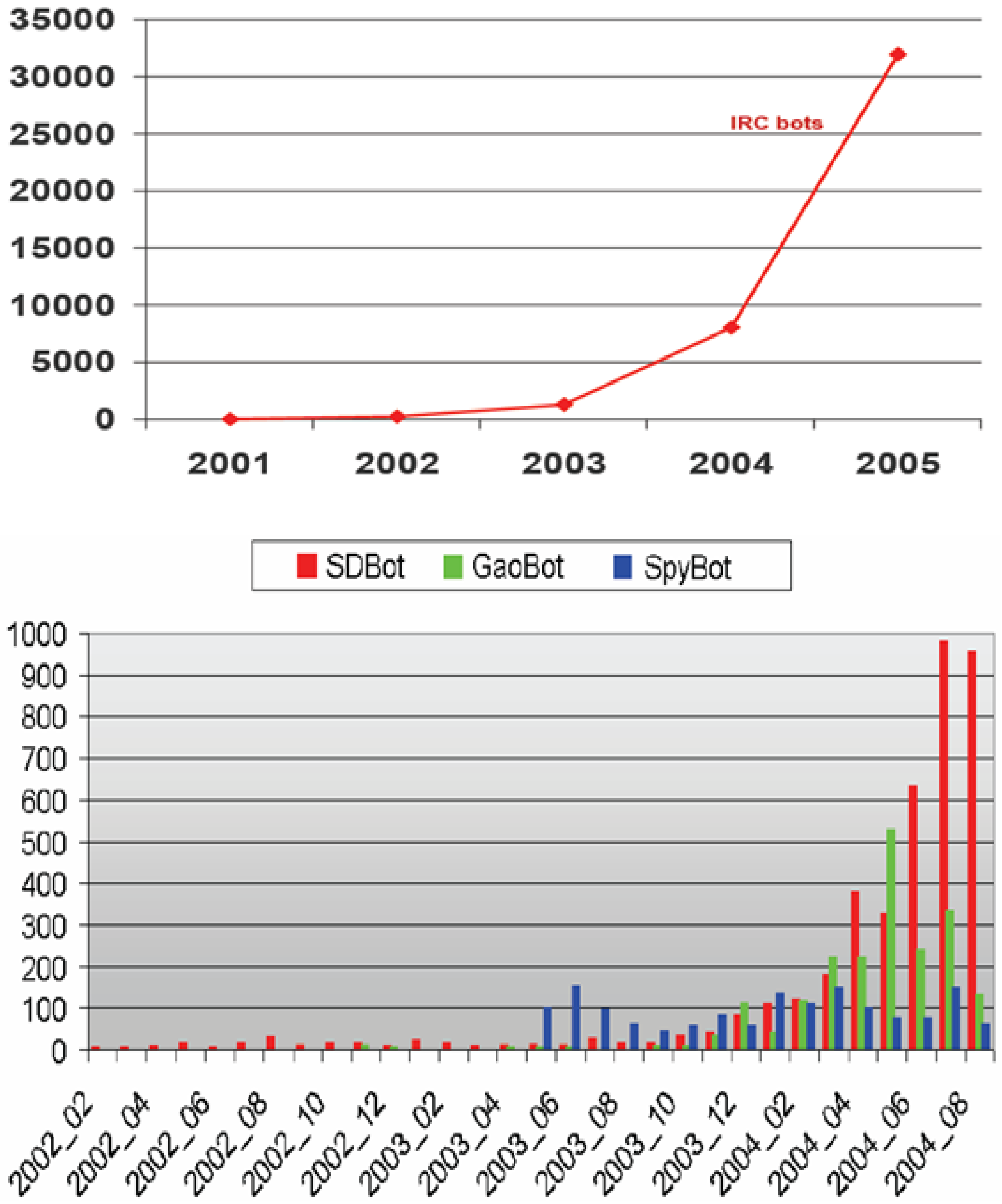

Figure 1.1: Exponential growth of IRC bots from year 2001. The above two figures are taken from [64].

150,000 new bots every day as shown in Figure 1.2.

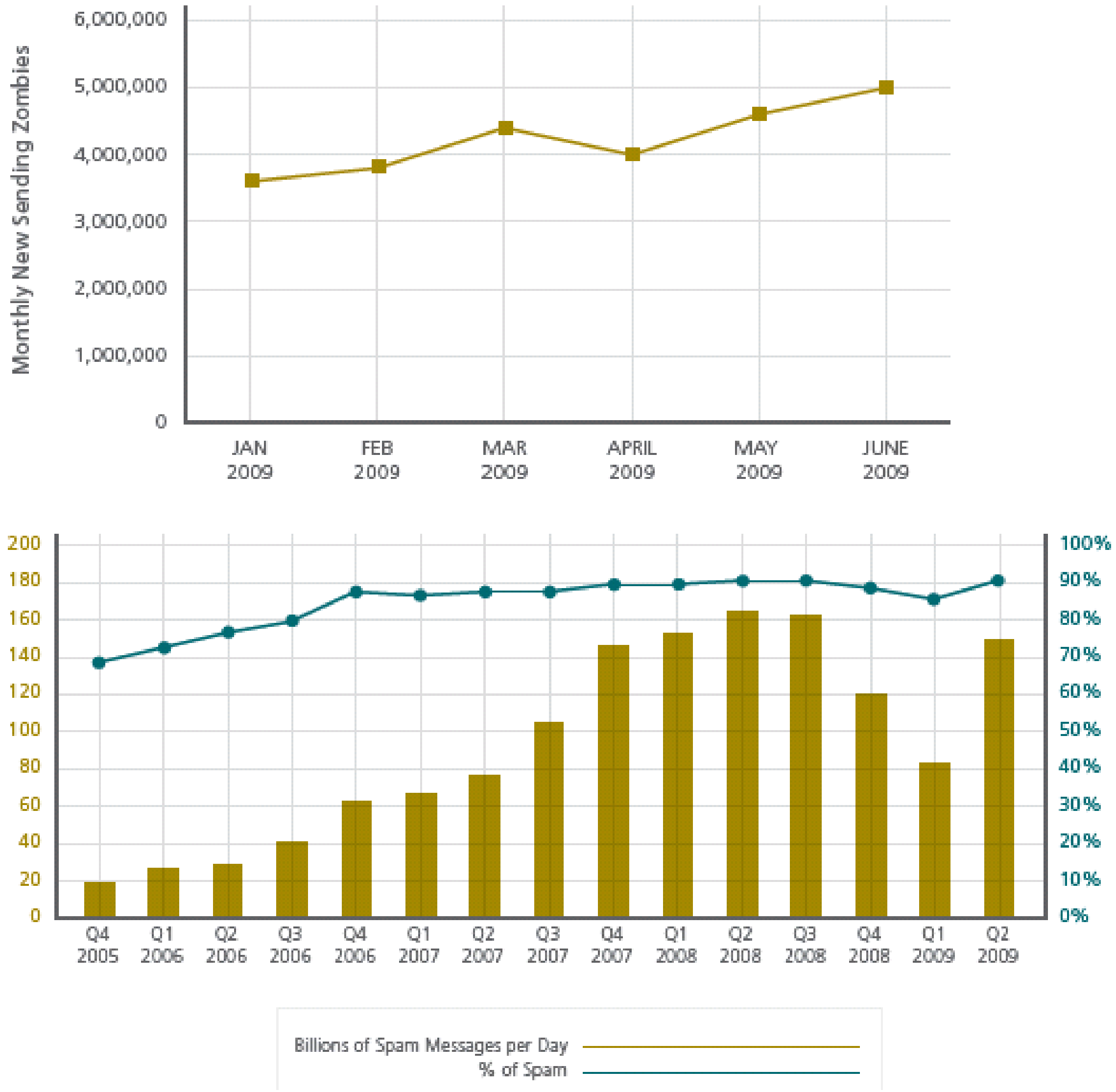

Figure 1.2: The top figure represents new bots sending spam by month while the bottom figure represents the Global Spam Volumes and Spam as a Percentage of All Mail.

As a result of the upcoming threats posed by bots, we decided to investigate this area in more detail and come up with new ideas of collecting botnet/bots data and detect their behaviour. Since most of the existing bots target Windows operating systems, our first step is to develop a program which monitors the interaction of these bots in Windows environment. In addition, our model is expected to detect these bots without prior knowledge of its signature as compared to the existing methods. Furthermore, the module should not depend on one type of behaviour in order to detect the malicious activity but should rather correlate different activities to enhance the detection mechanism. Moreover, the model should use an intelligent way of combining these behaviours to enhance the detection process. The module should be able to detect different types of malicious bots such as IRC bots, HTTP bots or Peer to Peer bots.

1.2 Aims and Scope

Many existing techniques use Intrusion detection systems in order to detect and prevent malicious activities on the network/systems. Intrusion detection is the ability of the system to detect and identify unauthorized activity by monitoring inbound network traffic. Intrusion detection reports malicious activity on the system to the security staff. In this thesis, we focus on extrusion detection rather than an intrusion detection. Extrusion detection is the ability of the system to recognize malicious activity by inspecting the outbound traffic [14]. Our aim is not to prevent the attack using well know techniques such as filtering known signatures but rather to detect a bot on a compromised host with a minimum number of false alarms without using a pre-defined set of signatures. The main aim of this work is to detect the existence of botnets on multiple systems and an individual bot on a single system respectively by correlating different bot's behaviours in Windows environment. This task is achieved by monitoring selected Application Programming Interface (API) function calls executed by the running processes on the systems within a specified time window. In addition, different correlation algorithms are examined in order to enhance the detection process. Another aim is to verify that the correlation algorithms can detect different kind of bots which use different command and communication protocols such IRC bots and P2P bots.

To achieve our aims, some of the research questions are presented:

Hypotheses

- *Do we use Intrusion or Extrusion Detection*: Which system is more suitable for bot detection, intrusion detection systems or extrusion detection systems and why? Our assumption is that the bots are already installed on the victims' hosts, through an accidental opening of emails, which contains 'trojan horse' or by visiting malicious websites and downloading malicious programs. In such a case, we are not attempting to prevent the initial bot infection but to limit its activities whilst on a host machine. Therefore, we use an extrusion detection system since our main focus is to detect botnets/bots and this could be a better solution for this task.

- *Detection method and Data Collection*: How are we going to detect botnets/bots? Are we using signature-based detection or anomaly-based detection? The method that we have used in this work is to collect our data by monitoring and intercepting specified API function calls executed by processes (discussed in Chapter 3 which represent bot's activities and then correlate these activities within a specified time period. This method is a behavioural-based detection method and no signatures are needed in order to detect these bots. Does this method achieve the required task?

- *One activity or different activities*: Does one activity (e.g. bot implemented with keylogging functionality discussed in Chapter 3) represent enough evidence to detect the existence of bots? Do we need to monitor different activities and what are they? Are the activities related? If yes, how do we correlate these activities?

  Does the process of correlation increase the detection performance and reduce false alarms?

- *Botnet Detection*: Are we interested in detecting botnet? How difficult is it to detect botnets? How many hosts to monitor? Can a simple correlation algorithm detect the botnet since they exhibit similar actions? What are the advantages and disadvantages of using such a technique? Can we detect botnet if we have different types of bots, each acting in a different manner? Do we need a more intelligent correlation algorithm? These types of questions will be addressed in Chapter 4.

- *Bots Detection*: Why we are interested in bots detection? Can we detect a single bot running on the infected host? What kind of technique do we use for bots detection. What are the most interesting behavioural actions of bots? Can we monitor and correlate these actions? How to correlate these actions? Is a simple correlation algorithm suitable for correlating different actions and thus detecting malicious activity? What are the consequences of using a simple correlation method? Can a more intelligent correlation algorithm enhance the detection performance in comparison to using a simple correlation method? Do we need a training phase when using such an algorithm to detect bots? What are the requirements which increase the detection performance? What types of input data do we need to change the input data every time new bots appear? Can our framework detect bots with low false positives and high true positives? This aim will be explored in Chapter 5 and 6 respectively.

- *Command and Control C&C Structure*: Can our framework detect different types of command and control structures other than IRC bots such as P2P bots? This issue will be addressed in Chapter 7.

- *Evaluation*: How good our algorithm is in comparison to the existing methods? How resilient the algorithm is to the change in bot‘s behaviour? What will happen if the behaviour or implementation of a bot changes? These questions will be studied and evaluated in Chapter 7.

1.3 Contribution

We can summarise our contribution on botnet/bot detection area according to the following points:

- Previous bots detection techniques rely on analysing network traffic looking for bots signatures. Other techniques use anomaly-based detection in order to detect existing bots. Other techniques use behavioural-based detection to detect these bots by monitoring system calls or API function calls. Because our approach is based on behavioural-based detection, we focus on monitoring API function calls in Windows environment. In contrast to existing approach, we target specific API function calls executed by the processes in order to detect malicious activities. Rather than using existing monitoring software which monitors many function calls, we developed a tool to monitor the most frequent function calls used by most of the common bots to perform malicious activities. These function calls represent (1) network activity: the communication functions which are needed to contact the botmaster or other bots, (2) process activity: file access function calls in order to store the data and finally (3) user activity: the keyboard status function calls to detect the keylogging activities on the system. Monitoring only the most important function calls used by bots can reduce the size of the data and the time for processing the collected data of function calls.

- Based on that, we have developed a framework for detecting bots which is ideal for both detecting botnet and an individual bot.

  1. for the botnet detection, we notice that many existing techniques use network-based techniques and look for botnet signatures in order to detect the botnet on the network. These techniques have many issues such as they may fail to detect new bots when the signatures are changed or they only have been developed to detect one type of bots. Little research has been conducted on host-based botnet detection. Because bots exhibit similar activities when they receive commands from their master, we implement a host-based botnet detection by monitoring the correlation of different activities from different sources within a specified time window. In comparison to existing techniques, no bot‘s signatures are being analysed to perform the detection. In addition, this technique can detect different types of bots regarding of their communication protocols.

2. for the individual bots detection, we notice that most of the existing techniques detect botnet rather than individual bots. To the best of our knowledge, only one or two methods were used for a host-based bots detection. As a result, we focus on detecting an individual bot running on the system and find a simple and effective solution in terms of detection performance and analysing the collected data and the time for generating the results. We start with a simple correlation technique represented by a Spearman's rank correlation (SRC) algorithm which is used to correlate different activities generated by processes and find the relationship between them. This method only states if there is malicious activity on the system but does not specify which process causes this malicious activity.

3. By applying the SRC algorithm, we noticed that this algorithm can be defeated in many ways such as allowing the bot to perform actions in different time periods and thus defeating the correlation between activities. In order to solve the problems generated by the SRC algorithm, an intelligent way of correlating different activities is used to enhance the detection mechanism. This is represented by a Dendritic Cell Algorithm (DCA). The DCA performs multi-sensor data fusion on a set of input activities, and this information is correlated with potentially anomalous 'suspect entities'. One advantage of using such algorithm is that the correlation between multiple data can be in different time periods. In addition, the DCA consists of multiple agents, the cells, and each cell has its own judgment on the active process based on the information it received. This process adds a robustness to the system because the decision is made by the majority of agents instead of one agent. In comparison to the SRC and other existing host-based bots detection techniques, the DCA does not only specify if there is a malicious activity on the system but it also indicates the malicious processes. In this algorithm, we modify the old anomaly value which is the mature context antigen value (MCAV), used in DCA, and present the MCAV antigen coefficient (MAC) to increase the detection sensitivity

and reduce the false positive alarms generated by the MCAV value.

- Our framework, either botnet detection or bots detection, does not depend on one type of botnet command and control structure. The framework can detect IRC bots in addition to the Peer-to-Peer bots. Although not tested with bots that use a hybrid structure, we expect that our algorithm can also detect these kinds of bots.

- A comparison between our framework and some of the existing methods is presented to measure the performance of our framework. We have also developed a non-DCA algorithm for detecting bots and compare its detection performance to the DCA. The results show that the DCA achieves a better detection performance than the developed algorithm.

- We also examine how the detection algorithm is resilient to the changes in bots. This step can show if the DCA is applicable for new types of bots which exhibit different behaviours than the current one.

- The final step that we have performed is to prove that the DCA can detect malicious processes other than bots and can be applied in many security areas.

A detailed description of these techniques and their advantages and disadvantages will be addressed in the Methodology chapter.

1.4 Thesis Outlines

This thesis is structured as follows. First, we start with the literature review chapter by defining different kinds of malicious softwares (malwares) that can affect our systems and networks. These malicious softwares include viruses, worms, rootkits and bots. Second, we present different ways of classifying intrusion detection techniques such as network-based vs. host-based intrusion detection techniques and signature-based vs. anomaly-based intrusion detection techniques. Third, we move more specifically toward the topic of this thesis by presenting the definition of bots, their history, the life cycle of bots and the command and control structure which is used by attackers to remotely control the bots. We also present the definition of botnets, their types, how they operate and communicate, scale of botnets, botnet architectures and the malicious use of botnets. In addition, we present some examples of existing bots which include some peer-to-peer bots and future bots. Fourth, we discuss some of the previous bots/botnet detection techniques and list their advantages and disadvantages. Finally we present a critique of the existing algorithm techniques and compare and contrast them with our work.

Because we are targeting bots which operate on windows environment, chapter three starts by giving a brief introduction on windows architecture and windows Application Programming Interface (API) function calls. Second, in order to collect the data, we use the hooking techniques. These techniques are based on intercepting API function calls which are explained in details in this chapter. Third, we present an abstract view of our framework for detecting botnets and an individual bot on a system. Fourth, we introduce our algorithm for detecting botnets by correlating log files in detail. Fifth, we also introduce our algorithm for detecting an individual bot on a system using different correlation methods such as the Spearman's rank correlation algorithm (SRC) and the Dendritic Cell Algorithm (DCA) and discuss their inputs, analysis, outputs and assumptions being used while implementing these algorithms and finally the strengths and weaknesses of each algorithm. We also present how the DCA is capable of detecting different bots, which use different command and control mechanism mainly P2P protocol. Finally, we summarize and conclude the chapter.

In chapter four, we present the design and the implementation of the algorithm for detecting botnets by correlating log files from different sources. We discuss different experiments that we have conducted to achieve this task and show the results of using such an algorithm. We finally summarize and conclude our findings.

In chapter five, we present the design and the implementation of Spearman's rank correlation (SRC) algorithm used for detecting individual bots running on a host based on correlation of different activities, mainly keylogging activity. In Chapter six, we use an advanced correlation algorithm, the Dendritic Cell Algorithm (DCA), for correlating different behaviours of an individual bot on the system and compare the results obtained with SRC algorithm. We also have conducted different experiments to achieve this task. Furthermore, we analyse and discuss the results of both the algorithms and we summarize and conclude our findings.

We further extended the scope of our research by presenting the detection of peer to peer (P2P) bots which is explained in chapter seven. We discuss how to detect such kinds of bots by using the DCA algorithm. Different experiments have been conducted to verify that the DCA can detect such bots. We analyse and discuss the results of detecting such bots. We evaluate the DCA algorithm with other existing techniques and develop a new algorithm to compare our results. We also show how resilient the DCA is to changes in the bot and what happens if a bot changes its behaviour? We also examine if the DCA is suitable for detecting malicious software other than bots. Finally we present our conclusion.

The final Chapter (Chapter eight) is about our conclusions that we have derived from this work, the critical assessment of the algorithms which describes how well the Spearman's rank correlation algorithm and the DCA algorithm performed and what needs to be done in the future in order to improve the algorithms for botnets/bots detection.

1.5 Summary

Botnets and Bots pose severe threats to our networks and systems. In order to reduce the impact of this threat, we need to find a way of detecting these bots to reduce its negative impacts. In this work, we try to detect these bots by correlating different activities within a specified time period.

Chapter 2

Literature Review

2.1 Introduction

Over the last decade, Internet is widely used by many people all over the world due to its availability and low cost. As more and more people use the Internet, network security as well as system security, become important elements to the users. This is because the Internet faces different types of threats from the attackers using malicious softwares (Malwares). Malware is a script or macro which is defined as a software used to breach a computer system's security policy [133]. Malwares are usually classified based on their activities. For example, a virus is a piece of malicious software which spreads from one computer to another by copying itself into files and then copies itself to the victim machines. Another example is a worm which is also a malicious code that acts as a virus but rather than transmitting to files, it copies itself via networks. These malwares are used as a mean of vandalism to attack victim machines. Recently, many attackers shifted their attacking strategy from vandalism through the use of viruses and worms to financial gains. To achieve this goal, the attackers have to have a large number of compromised machines all over the world to attack a single entity. These compromised machines are controlled remotely by a single or a group of hackers and called bots. The use of bots focuses on establishing Distributed Denial of Service (DDoS), extortion, spam, phishing and identity theft. In order to understand these types of threats, it is useful to distinguish between worms, viruses and bots. In the next section, we will present different types of malware and intrusion detection systems. Our focus will be on bots and botnets, what are they, how they operate, examples of these bots and existing methods to detect these bots.

2.2 Malicious Codes

Internet faces different types of attacks generated by malicious softwares - malwares. In order to detect these attacks, researchers should differentiate between the behaviour of the malwares. In this section, we will present different types of malwares including viruses, worms, rootkits and bots and explain their functionalities.

2.2.1 Viruses

A virus consists of two parts, insertion code and the payload. The insertion code adds a copy of itself to other executable programs to infect other programs while the payload involves the malicious activities which may produce serious damage to the files or system [5][133][139]. A virus requires manual interaction to be activated, and cannot run independently on its own power [114]. Viruses can propagate through file-to-file replication [165]. There are some techniques which a virus creator takes into account before designing a virus. These techniques include propagation, which allows the virus to transfer from one file to another and infection method, which relates to files selection to infect, code placement into victim files and the execution strategy. Some viruses adapt encryption techniques on their payload in order to evade detection. Other viruses use polymorphism techniques (having different types of virus code to achieve the same goal) to hide their activities from signature scanners.

2.2.2 Worms

Spafford [139] defines a worm as a piece of malicious program that can run by itself and can propagate a fully working version of itself to other machines through the network without infecting files. The word worm is derived from the word 'tapeworm' used to describe a program in John Brunner's 1972 novel [107], the Shockwave Rider, which lives inside the host and uses its resources to maintain itself and spread to other machines. This cycle is repeated to increase the number of infected hosts [114]. A worm is like a virus but consists of three parts. The first part is designed to search for vulnerable targets based on information found on the currently infected host or based on port scanning. The second part is responsible for transferring the worm code from one host to another, and the final part is responsible for code execution. Most of the worms are designed to consume network resources rather than host damaging by consuming bandwidth or establishing distributed denial of service by exploiting the targets [133].

2.2.3 RootKits

A rootkit [133][68][175] is a collection of tools that is used by the attacker to gain administrator access privileges on a host. Rootkit allows the attacker to sniff network traffic to gather information about the system or networks, to apply keystrokes logging to retrieve sensitive information typed by the user such as credit card numbers, passwords or personal information, or to hide the attacker's presence as well as to hide the malicious programs from rootkit scanners. The rootkit can act as a back door which allows the attacker to gain access to the victim's machine at any time.

2.2.4 Bots

A bot is a malicious piece of software that can be installed on a user machine without his/her knowledge. Once it is executed, the bot connects to the IRC server and joins the channel specified by the attacker. These bots are controlled remotely by the attacker. Therefore, the main difference between the worm and the bot is that the bot offers a remote controlled channel to the attacker. Today's bots combine features of viruses and worms. For example, they propagate like worms through network shares, file-sharing platforms, peer-to-peer networks, backdoors left by previous worms and/or exploit vulnerabilities. They also can hide their existence like viruses using rootkits. Bots can communicate with others using different types of protocols such as IRC, HTTP or Peer-to-Peer protocols.

2.3 Intrusion Detection Classification

Intrusion Detection System is the process of gathering and analysing information from different sources within the networks or systems for detecting events to track security violations or attempts [78][95]. These events include network attacks against vulnerable services or host-based attacks such as unauthorized access. Intrusion detection systems usually store a database of well known attack signatures and compare the activities that monitor within the networks or systems with the signatures they have. If a match is found, they generate different types of alerts based on the type of attack. These alerts are then forwarded to the system administrator who tries to discover and process the alerts. The main goal of intrusion detection systems is to detect and then deflect unauthorized attempts to the system or network. Intrusion detection systems can be classified on the basis of activities, traffic or systems they monitor. They can be divided into network-based, host-based or application-based intrusion detection systems. Intrusion detection can also be classified based on event analysis such as a signature-based (misuse) intrusion detection or anomaly-based intrusion detection. In this section, we will describe different types of intrusion detection systems based on their classification.

2.3.1 Network-based IDS vs Host-based IDS

Network-based intrusion detection systems (NIDS) monitor the entire network rather than individual systems looking for the attack signature without interfering with network operation. Thus, a single alert will be produced rather than multi-alerts from each host. Network-based IDS can be a reactive intrusion detection system that can respond to the given attack without alerting the system administrators. Such actions include reconfiguring the network or blocking malicious data. One problem of having network-based IDS is that they may not be able to monitor and analyse all traffic on high-speed networks. This is because it is very difficult to capture all packets when data processing can not cope with the speed and throughput of networks. The packets which are not analysed on time can be dropped and these packets may contain the attack signature [176]. In addition, they may suffer from false positives by identifying an attack which is in reality normal behaviour and false negatives by missing an intrusion attempt. Furthermore, if the packets are encrypted, network-based IDS may not able to analyse these traffic.

On the other hand, host-based intrusion detection systems (HIDS) monitor and analyse system network traffic, activities and attempts to access operating and file systems. They can also monitor abnormal activities of processes and search through log files and alert the users for possible attacks. Host-based IDS can use host-based encryption services to examine the content of encrypted traffic, thus protecting the entire host. One of the disadvantages of using host-based IDS is that they need to be installed in every single host. If the host is compromised, the attacker can disable host-based IDS. Moreover, host-based IDS consume a lot of host resources such as processing time, memory and storage. Anti-virus tools and system calls-based monitoring are examples of host-based detection [38]

An application- based IDS monitor's events that happen on some applications and detect the attack, based on log file analysis. They also can examine the content of encrypted packets by using application-based encryption services. The disadvantage of using application-based IDS is that they are more prone to attacks and consume host resources [4][32].

2.3.2 Signature-based detection vs Anomaly Detection

Intrusion Detection Systems (IDS) for detecting bots in networks can be classified into two general categories: Signature-based Detection and Anomaly-based Detection. Signature-based detection is based on defining malicious patterns that the system has to detect [96]. For example, an Intrusion Detection System (IDS) that analyses web server traffic might look for the string 'phf' as an indication of a CGI attack [10]. Signature-based detection suffers from the problem that it cannot detect new malicious behaviours. It requires that a signature of each attack be known in order to find attacks. If the signature is not known, then signature-based detection will fail to detect these malicious attacks. Examples of signature-based intrusion detection are Snort [138] and Bro [119] which consist of a large number of previous attack signatures on their database to detect the attack.

In contrast to signature-based detection, anomaly detection differs by constructing a profile of normal behaviours or activities on the network, and then looking for activities that do not fit the normal profile [4][32][96]. Since not all the abnormal activities in the network are suspicious, anomaly detection has the problem of raising false alarms when it encounters normal traffic that it has not seen before. However, anomaly detection has an important advantage that it can be used to detect new malicious activities for which there is no known signature. For this reason, our focus is on how to develop anomaly detection techniques for bots.

2.4 Bots and Botnets

2.4.1 Bots History

IRC history

IRC stands for Internet Relay Chat, which provides a way of communication with connected users in a real time [84][113][159][168]. It is mainly designed for group (many-to-many) communication in discussion forums called channels. In addition, IRC allows uni-cast communication [118]. The user can monitor a conversation between multiple users and can participate in the conversation. IRC was created in late August 1988 by Jarkko Oikarinen to replace a program called MUT (Multi-User Talk) on a BBS called OuluBox in Finland and to allow a maximum of 100 users to communicate concurrently [84][118][168]. In 1993, the first IRC protocol was defined by RFC1459. Later on, it was updated to include RFC2810, RFC2811, RFC2812, and RFC2813 [79].

IRC Operations

Once a user connects to the IRC server, s/he can join a channel where other users are already there as shown in Figure 2.1. If s/he is the first user who joins the channel, s/he will be the channel operator. Any user submits a message to the server publicly, the other users can see her/his message on the channel [84][113][26].

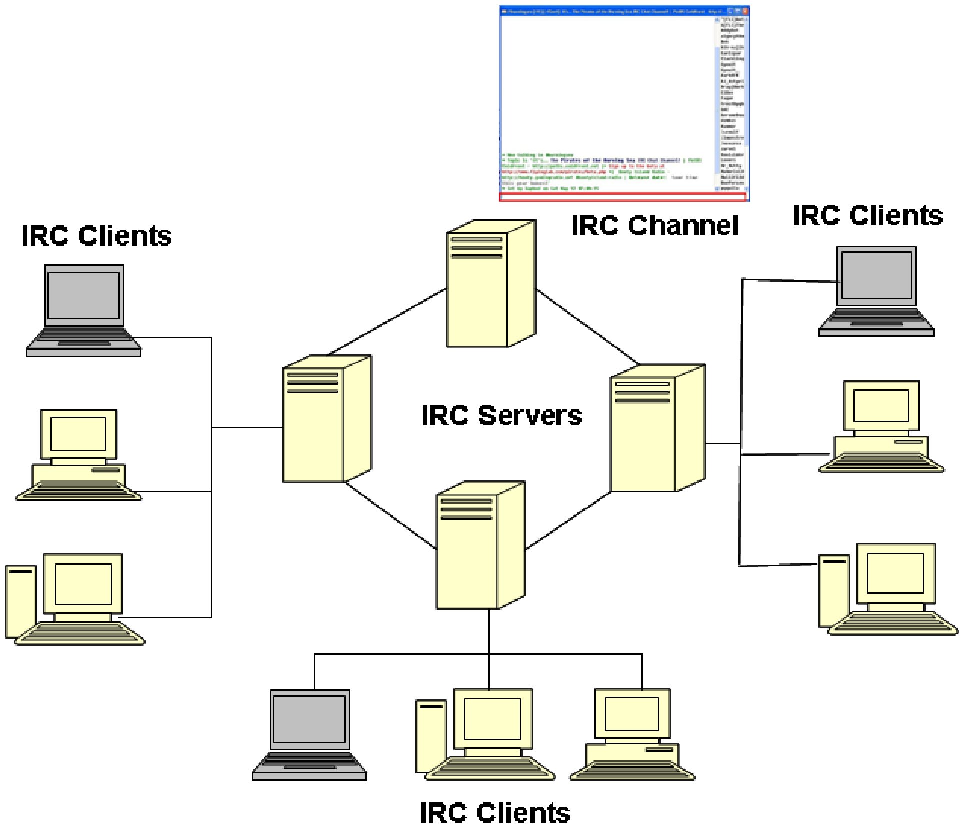

FIGURE 2.1: Basic IRC Operation

IRC has multiple channels. Therefore, if any user wishes to join a channel s/he should have the IRC client, know the IP server and also should know the channel name. A user can connect to IRC server through predefined ports (6660-7000/tcp). The most common port of the IRC server is 6667/tcp. Each channel has one or more operators to manage the channel. The channel operator's privileges can be obtained by one of the following methods. The first method is that the user creates that channel and becomes its operator. The second method to obtain operator privileges is through an Approved Channel Operator (AOP) list. In addition to sharing text messages between users on the channel, IRC has other functionalities. For example, IRC can allow file transfer between users, execute peer-to-peer capabilities, and run an automated program, termed 'a bot', to monitor IRC channels. Moreover, IRC has many commands that can be used by users. In this section, we will define the most common commands, which are used by users.

- NICK and USER: are used to label a user and user's host equivalent to an ID.
- PASS: set or send a password.
- JOIN: enter a channel, often secret.
- MODE: modify channel settings (eg. invisible).
- PING and PONG: maintain the connection to IRC server.
- PRIVMSG: send a message to a channel or user.
- DCC SEND: transfer files from one user to another.

IRC and Bots Relationship

In the 1990's, users developed new tools such as logging channel statistics, running games, and providing a mechanism for file distribution to automate certain tasks and to defend them against attacks [159]. These automated tools are termed bots. The IRC bot is a non-human client that has been programmed to respond to various events and to help maintain the channel operator status on a particular channel [168][84]. A bot can be used to manage operations such as granting operator status to recognised users. Other users have created attack tools to kill channels and remove users from these channels and to fight back. These tools are called malicious bots [118]. By the end of 1990's, bots were programmed to do some malicious activities. One of these activities was to establish distributed denial of service (DDoS) attacks against IRC servers. This was due to the loss of operator privileges on IRC channels when the server crashes or disconnects, and another member of the channel is assigned as a new operator when the server is up. As a result, people started to attack IRC servers so that they acquire the privileged operator status in a given channel. DDoS is established when many bots are gathered together, which form a botnet, under a control of their master.

According to Trend Micro [165], the first bot to appear was in 1999 and was named PrettyPark. PrettyPark bot implements a way of controlling malicious program remotely using IRC networks. The PrettyPark bot has a limited set of functionalities such as the ability to connect to the IRC server, steal basic system information, login names, nicknames and email addresses. More advance bots were created after PrettyPark and have more sophisticated functionalities.

Most of the bots used, to attack IRC did not use IRC to communicate. At the start of 2000, many sites such as Yahoo, eBay, Amazon, and CNN have been attacked by a Canadian hacker [118]. During 2002, new communication protocols for bots/botnets were developed. From 2003, bots used different techniques to spread. These techniques include exploiting vulnerabilities, buffer overflow, droppers (programs that are used to install malicious software in a target host), and dictionary attack.

Bots should be available and active most of the time on the channel, therefore, they are run from high availability servers with reliable and fast Internet connections. Furthermore, they are controlled from a shell account on the server [84]. Nowadays, bots can be delivered by most of the ways that current malware can. Today, bots are used for extortion [159], spam and phishing, identity theft, and malware seeding.

2.4.2 Bot's Life Cycle

Initial Infection and Propagation

Bots can spread and propagate only if there are existing vulnerable hosts. The attacker appends the bot by gaining access to the victim's host. The victim's host is infected with a bot by exploiting some operating system or application vulnerability [126], through malicious websites, by opening emails that contain malicious data, or downloading malicious payload from P2P networks [177][113]. The malicious payload is polymorphic and transparent to the user. The user has no clue that his computer is being used for malicious purposes. Once a bot is installed on a victim's machine, it changes the system configuration to start each time the system boots [126]. The bot might have the functionality to spread itself by sending out more emails or scanning more computers [113], thus creating a botnet. If an attack is detected, it can only be traced back to the source, not to a botmaster.

Command and Control server (C and C)- Remote Control Channel

Once the bot is installed, it has an instruction to connect to command and control (C&C) server to receive commands [177][113]. Command and control allows control of compromised machines from one centralised system, typically through IRC channel. The bot (i.e. infected machine) connects infected machines to the IRC server with a randomly generated nickname [126]. Then, the bot joins the attacker's channel with a predefined password, and remains dormant waiting for the command from the botmaster. The botmaster authenticates the identity of the bots using a password, ensuring that the bots cannot be controlled by others. Once authenticated, the botmaster issues commands in the channel and all bots react and respond to the commands [177][113].

Accepting Instructions

The botmaster logs into the IRC server and starts issuing commands. Commands are sent to infected machines via the controllers. These commands can include instructions such as downloading additional payload to bots. In addition, the botmaster can transfer files to/from bots, perform DoS attack, or use them for other malicious activities.

Spread to additional machines

Botmaster uses C&C to exploit new vulnerabilities of other systems which allow bots to spread to other machines through the local networks, often bypassing firewall and IDS.

2.4.3 Botnets

Introduction to Botnets

Bots, which have infected a large number of vulnerable computers across the world, forms a botnet. Botnets have become the current threat to the most interconnected systems [113]. According to Bacher et al. [9], several thousand bots can take down any website or network instantly, thus, making botnets loaded and powerful weapons. A *bot*, a term derived from the word *RoBot*, is a piece of computer code, which runs on a client system and connects to the IRC server to join a specific channel. A bot can automatically execute predefined commands such as updating itself or installing new malicious software [118][113]. A *botnet* is a network of compromised machines infected with malicious programs (bots, zombies) that can be remotely controlled by an attacker through a Command and Control (C&C) infrastructure on IRC channel or Peer-to-Peer (P2P) network [118][113][76]. Botnets often consist of thousands of compromised machines, which enable the attacker to cause a serious damage. Botnets are used for DDoS attacks against the target machine to suspend services by utilizing the available bandwidth. To establish DDoS attacks, the botmaster instructs an army of compromised machines (bots) to attack the victim's machine simultaneously [118][113].

Type of Botnets

There are many types of botnet; hub-leaf and channel. The Hub-leaf botnet is made by installing two bots on the victim's machine. The first bot is configured as a hub while the other bot is configured as a leaf. Additional bots can be configured as leaves, which connect to the hub bot. The resulting connection will form a star architecture. Hub-leaf botnets do not typically communicate through an IRC, but rather on configurable ports. Another type of botnet is called a channel botnet as shown in Figure 2.2, which allows bots to communicate through an IRC channel. Once a bot is configured on a victim's machine, it joins a predefined channel. The botmaster issues commands by posting messages to the IRC server. Bots read these messages, interpret and react to them. Another type of bot includes AOL bot [177], which logs on to a set of AOL servers to receive commands. In addition, P2P bot uses peer-to-peer file sharing applications to spread, or to communicate with other bots [159].

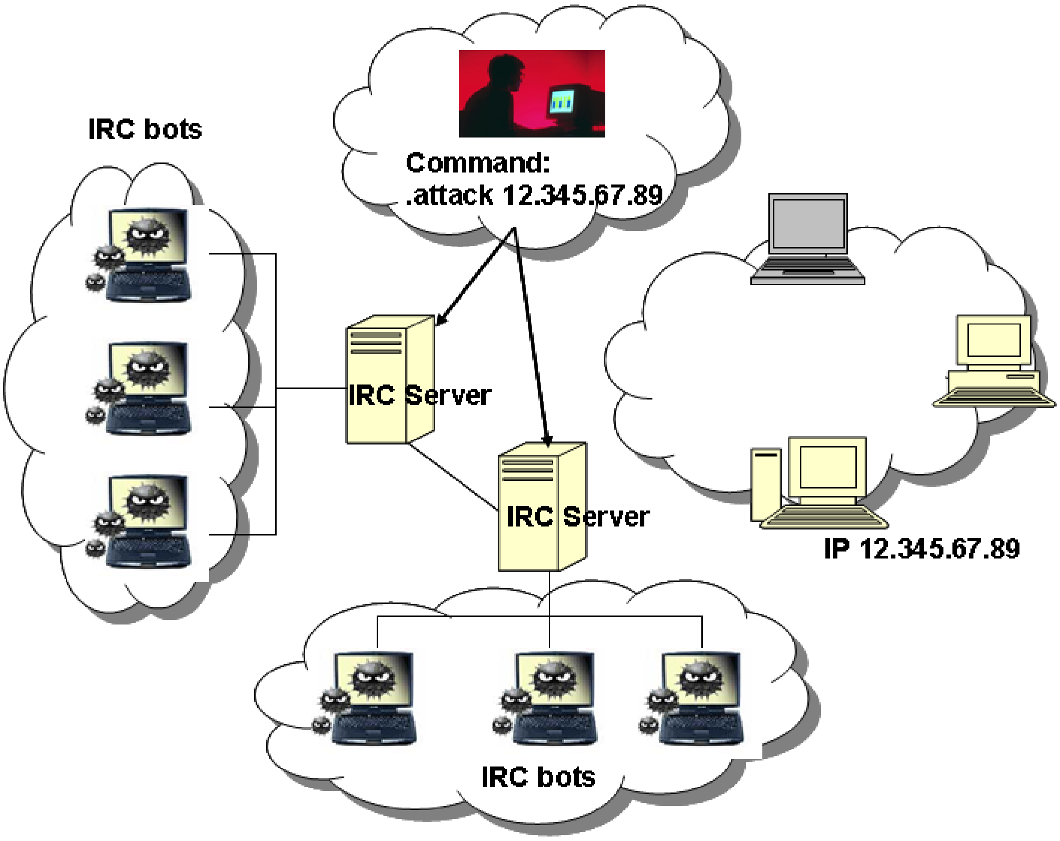

FIGURE 2.2: Structure of an IRC Channel Botnet.

Techniques for Creating Botnets

In the past, the intention of malicious users and hackers was to attack systems especially IRC servers and control channels. They also used botnets for communication, resource sharing, and curiosity. Nowadays, malicious users have moved a step forward. With the rapid growth of e-commerce and online financial transactions, the aim of malicious users, has shifted from curiosity to financial gain [76]. To achieve this goal, a powerful weapon is needed. Therefore, botnets were used to perform this task.

One way the attackers can create a botnet is to look for vulnerable systems to exploit. Vulnerable systems are carefully selected rather than randomly. For example, attackers are targeting broadband connections to maintain the availability and the capacity. These two factors give the attacker powerful botnets. Attacker's are also attracted by the high speed connectivity available on systems at universities. These systems give the attackers fast connection, the storage capacities they need, and less time to attack them because they are poorly secured. Furthermore, the attackers target sensitive military and government systems which have lots of valuable information, which can financially benefit them. In addition to vulnerability exploitation, attackers use social engineers to compromise vulnerable systems, thus, creating their botnets. Social engineering is a way of convincing naive users to take actions s/he would not otherwise take. Email, web browser, and instant messaging applications are ways of applying the social engineering attack.

Another way of creating botnets is by stealing them from another bots herder, known as 'hijacking' botnets. Most bots come with the functionality of sniffing the network. If an attacker notices that there is a competing botnet communication, he will try to hijack that botnet. In addition to hijacking other bot- masters, the attacker can advertise her/his botnet for sale or rent. The Botnets market is very competitive and whoever wants to rent a botnet, has to pay well.

How Botnets Work and Communicate

Most bots are controlled via IRC on ports 6660-7000. The most common port that IRC servers use is port 6667. Once a bot is setup on a victim's machine, it connects back to the IRC server, which runs a specific channel for bots and botmaster to log into. The botmaster will try to hide his botnet from the IRC server owner/admin by using certain mode commands [118]. In addition, botnets use dynamic Domain Name Server (DNS) from providers that offer dynamic DNS services. There are two reasons for using dynamic DNS services. Firstly, it makes the process of tracking botnets more difficult. Secondly, these services when configured are setup with a very short Time to Live (TTL). If the botnet server gets disconnected, the botnet will be lost for a short time until a new IRC server is specified or the original server comes back on-line with a new IP address [40][118][159]. Inside the channel, the botmaster will issue some instructions for bots. The first command may allow bots to search for other victims, thus, propagating to other networks. There are many methods for propagation. These methods include:

- Vulnerabilities exploitation: The attacker may combine different vulnerabilities to gain control of computers in case if one vulnerability cannot provide the level of access desired. Some of the common vulnerabilities which help in spreading bots are:
    1. Remote Procedure Call - RPC
    2. Distributed Component Object Model - DCOM
    3. Local Security Authority Subsystem Service - LSASS
    4. MSSQL
- Dictionary attack
- Server Message Block (SMB)
- Existing backdoors from the previous worm attack
- Download malicious files from a website via droppers (e.g. email)
- Peer-to-Peer files sharing.

In addition, the bot may be updated or new components can be installed.

Command and Control (C&C) Technologies

There are four technologies by which attackers can control their botnets. The first way to control botnets is through IRC Servers for Command and Control. IRC is the most commonly used C&C server by attackers. The advantage of using such technology is that it requires minimum effort and management. Attackers can easily issue commands that instruct bots to perform malicious activities in the channel. In addition, these channels can be private in which no other users can join [76]. Another way that the attacker can control the botnet is by using Web-Based (HTTP) Command and Control. Some bots are configured to instruct the compromised machines to connect to the web-based C&C server providing the machine's IP address, port, and machine identification. The compromised machine accesses a PHP script on a malicious website. In order to track and control a botnet, the attacker uses a web interface. The interface is used to send commands to the individual's compromised machine or to the whole botnet via HTTP responses. The third way to control the botnet is through using Peer-to-Peer (P2P) Command and Control. The IRC server C&C architecture is a centralised method for controlling botnets. If this server is down, the attacker could lose his/her botnet control. In addition, the botnet which uses a centralised server can be detected easily. In order to avoid detection when using IRC protocol, some attackers use P2P networks in order to control their botnets. P2P networks have no real server that can be shutdown to disable the botnet. The final way of controlling the botnet is through the DNS server Command and Control. In this way, the botmaster employ DNS tunneling to communicate with his/her bots [24]. Tunneling techniques are used to tunnel traffic in and out of the firewall and intrusion detection systems. The advantage of using the DNS server as a C&C mechanism is that the DNS server is very popular, used by anyone, and permitted by the majority of firewalls. In addition, the bots can use free DNS hosting services to query about the location of IRC servers which are controlled by the attackers.

2.4.4 Scale of Botnets

It is difficult to estimate the accurate size of botnets due to the distributed and anonymous nature of this problem [113]. According to NISCC [113] and Overton [118], UK has the largest zombie PC population over the world and more than one million connected PC's are zombies. There were more than 300,000 Internet connected zombie networks in 2004. Approximately, 25% of infected PC's are controlled by hackers. Broadband connection is responsible for the 93% increase in infected PC's in 2004. A recent paper by Bacher et al. [9] used honeynet over a period of four months to monitor botnets activities that have different size and structure. A honeynet is an entire network of systems which consists of multiple honeypots and firewalls [98][93]. Honeypot is a resource that is not meant to provide any service to legitimate users [13]. The only traffic that a honeypot receives is probe traffic or other malicious traffic. Thus, any traffic that is captured at the honeypot is assumed to be unauthorized and malicious, and can be analysed to identify any threatening activities [13]. According to Bacher et al. [9], they have observed and tracked more than 100 botnets during these four months. They have logged 226,585 unique IP addresses logging into one of the IRC botnet C&C channels [113][118]. The size of botnet ranged from several hundred to more than 50,000 zombies. They also observed 226 DDoS attacks against 99 unique targets. The logged bots are being ordered to update themselves or to download and run new malware. Other studies by Cooke [26] and Jahanian [80] show that botnet sizes range from several hundreds to few thousands infected hosts.

Even with these statistics, the actual size of each botnet is difficult to estimate due to a number of factors. These factors include simultaneous bots logging, having more than one bot on a single compromised host, and networks that block bots from connecting to the server. The size of botnets varies. Nowadays, botnet controllers prefer small, more manageable botnets, which are easier to control and protect, rather than having a botnet with thousands of bots. Actually, there is limited information about the geographical distribution of bots and botnet controllers [113]. It is hard to track back the malicious users who control the botnets. Although analyzing the attacks may reveal the location of compromised hosts, and identify the location of a control server, it is difficult to identify the location of bot controllers. The only fact that we can state about the size of botnets is that there are existing botnets of sufficient size that can successfully attack almost any Internet connected application [113].

2.4.5 Botnet Architectures

Different botnets use different topologies in order to communicate. In this section, we will describe these topologies in detail which include centralised, decentralised and hybrid topology.

Centralised

The centralised topology consists of a single point in which all bots receive commands and data from the botmaster. For example, IRC is a centralised C&C structure in which all the infected hosts try to connect to the IRC server and join the same channel. In a similar way, the botmaster connects to the IRC server and joins the same channel in which he starts to communicate with the bots by initiating different types of commands. The advantage of using this method is that the IRC network provides flexibility to control the bots and there is little latency. The disadvantage of using this method is that the botnet can easily be detected and can be lost or the command and control server will be disrupted or shutdown, thus, the botmaster will not be able to communicate with his bots [142][65][26][57].

HTTP-based represents another way of controlling bots using a command and control server, usually on port 80. In this method, the botmaster publishes his commands on the web server and the bots get these commands. The advantage of using this method is that HTTP traffic is widely used thus making botnet detection more difficult in comparison to the IRC server. Moreover, the IRC traffic is usually blocked by most firewalls but not HTTP traffic [142][65].

Decentralised

In addition to the centralised topology, botmasters create another way of controlling bots through peer-to-peer networks. In this method, the botmaster communicates with bots directly or the bots can connect to each other. All peer nodes are equal as they do not have a centralised point which can be shutdown to disable botnets. The advantage of using this method is that peer to peer communications are harder to detect than the centralised communications. In addition, if one bot is discovered, this will not have implications on the remaining bots [142][65][26].

Hybrid

In this case, each bot has its own peerlist to communicate with other bots. The botmaster will send the command to one or more bots and these bots will try to communicate with the bots on their list. If communication is established, the command will be forwarded to the other peers. The disadvantage of this method is that the delivery of the commands is not guaranteed and could be delayed. On the other hand, it will be hard to detect the origin of the botmaster by tracking the commands [142][65].

2.4.6 Malicious Activities

Originally, bots were used to monitor and manage IRC channels. As more and more users joined the channels, operators started to kick/ban misbehaving users from the channels. As a result, banned users started to develop malicious bots to attack IRC servers and channels. There are two primary malicious usages of botnets. First, the botmaster uses his/her botnet for file transfer and distributed file storage. The traffic on these private networks is hard to detect. Second, botnets are used to launch a DDoS attack against channel operators or webservers [113][76]. There are many ways by which botnets are used to deny services. One way is to allow bots to flood a site with too much traffic, thus consuming the application's connection bandwidth, which results in legitimate users unable to access the service [113]. In addition, systems that are not being directly attacked by DDoS, could be affected as large botnets could generate enough malicious traffic to overwhelm the ISP [113]. Diversity of botnets allows the attacker to launch DDoS attacks from different source addresses, which make it difficult to shutdown or filter. In addition, botnets are used for scanning, exploiting vulnerabilities, distributed computation, email spamming/bombing, and malware distribution [84][159]. Moreover, botnets can be used to steal system information and user data, terminate programs [177], and P2P propagation [164][159]. Nowadays, bots are designed to have a key logger, functions to grab cached passwords, and a function to capture a screenshot of infected machines. Some of these usages are explained in detail below.

- Distributed Denial of Services (DDoS): DDoS attack means attacking targets in order to exhaust system resources required to provide a service, slowing or stopping a legitimate request. This includes ICMP, ECHO, UDP, and SYN flooding [76].

- Email/Instant Message Spamming: Botnets are used to distribute unsolicited email (spam) and pop-up advertisements (adware) [40]. They can also be used to send a specified message to any open IM windows on the infected machines [159]. In September 2004, Spamhaus [140] estimated that 70% of all spam emails were distributed by botnets. In addition, with botnet delivering spam, it is difficult to detect the IP of malicious user's. Furthermore, a bot can hijack the victim's Internet connection, and distribute spam emails from there.

- Key Logging, Packet Sniffing: Computer systems are targeted by malicious users because they contain valuable information. Therefore, bots can be programmed to record the victim's key strokes. It can be coded to filter specific keywords of user personal or confidential information such as 'username', 'password', 'bank' or email contacts [40][76]. Moreover, bots can be programmed to sniff the packets that the victim sends out over the Internet, therefore, revealing some important personal and financial, or trade secrets information such as license keys for games, music, software, and even details of other botnets. The botmaster could command his/her bots to scan the victim's hard drive searching for files, or transfer files without the knowledge of the victim [159][113]. By using the above mentioned function, the botmaster can earn a profit by trading or selling the information that he gained.

- Vulnerability Scanning and Exploitation: Most recent bots contain some sort of vulnerability scanning functionality such as port scanner, or exploit scanner. Their code includes functions to automatically scan the local area network, or scanning random IP addresses. Botmasters use exploit codes which are a slightly modified version of publicly released code. Some of vulnerabilities that are used by bots are DCOM RPC, LSASS, MSSQL server, and UPnP NOTIFY buffer overflow [159][76].

- Process Killing: Most of the bots have a function of killing the anti-virus and other security products. The bot on the infected machine searches for a process that matches processes on the list that it carries and terminates that process [159].

- Anonymising Proxies: Botnets can be used to anonymise attacker activities such as hacking by providing unattributable 'proxies' for the malicious user [113][76]. In addition, bots can be used to strip any information about steps taken by the attacker. Thus, tracing the attacker back will be a hard task. Botnets provide not only anonymity, but also persistence. If one bot is detected, the attacker can quickly switch to another computer. Botnets can be updated remotely to add new functionalities, and support more targeted hacking activities [113].

- Download and Installation: Most bots have the ability that allows FTP, TFTP, HTTP download and execution of binaries [76]. This method can help the attacker to command the bots, update the malicious code, or download any files. By this means, the attacker can install other malware such as spyware, and adware [76].

- Storage Files: The botmaster can use compromised machines (zombies) as illegal storage spaces for software, films, and other illegal files. By using compromised machines, the botmaster has free storage space.

- Competition: Botnets are not only created for malicious purposes, but also are created to compete with other botnet creators. Sometimes botnets are rented or sold to an individual who will use them to do malicious stuff. After building, reasonable sized botnets, the botmasters advertise their botnets for sale or rent.

2.4.7 Peer to Peer Bots

One of the main problems that botmasters face when implementing botnets is losing control on their bots when using IRC protocols. Because the IRC network is a centralised structure, botmasters devised different ways of controlling and managing bots, which do not depend on centralised structure in case it is shutdown or discovered. As a result a Peer-to-Peer (P2P) command and control structure is used. Peer-to-Peer network is defined as a network in which the host in that network can

Table 2.1: The time-line of using P2P protocols and bots.

Date	Name	Type	Description
05-1999	Napster	P2P srv	First widely used central and P2P service
03-2000	Gnutella	P2P	Decentralised P2P protocol
09-2000	eDonkey	P2P	Used checksum directory lookup for file resources
03-2001	Fast Track	P2P	Use supernodes within the peer-to-peer architecture
07-2001	BitTorrent	P2P	Use Bandwidth currency to speed the download process
05-2003	WASTE	P2P	P2P protocol with RSA public key for small network
09-2003	Sinit	P2P bot	Bot using random scanning to find other hosts
11-2003	Kademlia	P2P	Use Distributed Hash Table for decentralised architecture
03-2004	Phatbot	P2P bot	Bot based on WASTE
03-2006	SpamThru	P2P bot	Bot using custom protocol for backup
04-2006	Nugache	P2P bot	Bot connecting to predefined hosts
01-2007	Peacomm	P2P bot	Bot based on Kademlia

act as a client and a server where there is no centralised point and any node can provide and retrieve information at the same time [57][71]. Peer-to-Peer command and control structure has many advantages over the centralised structure. One of these advantages is that it is hard to trace back or track the origin of the botmaster. In addition, if one bot is detected and shutdown, it will not affect other bots. Peer-to-Peer network is difficult to shutdown as compared to a centralised structure network. One of the earliest peer-to-peer worm was the Linux worm named Slapper [72]. Slapper uses a Peer-to-Peer algorithm in its source code. After infecting the machine, the worm tries to add the infected machine to the Peer-to-Peer network controlled by the attacker [120][74]. Since then, many botmasters started using Peer-to-Peer networks in order to control their bots. Table 2.1 explain the time line of using peer-to-peer protocols and bots [57].

Napster was one of the first programs that allowed the sharing of files between clients. The client connects to a centralised server where s/he uploads his/her files and searches for a specific file on other user hosts, and the server responds by indicating the position of that file. Then, the client connects directly to the indicated host position to retrieve that file. Napster is not entirely a Peer-to-Peer service because it contains a centralised point where the users connect to. Because many files were illegally traded, Napster was shutdown and stopped servicing. Motivated by Napster, people thought of implementing a complete Peer-to-Peer network and Gnutella was introduced. Gnutella is a complete Peer-to-Peer service [157] for distributed search and every peer in this model is both client and server. A more efficient way of searching for information in Peer-to-Peer network is based on distributed hash table (DHT) and used by Kademlia algorithm. The Kademlia algorithm provides an efficient method to find values for a given key. Every node is assigned a 160 bit as a unique identifier (ID). To publish and find a $< key, value >$, where a key is a generated hash function for a given file and a value is an IP address of the file location, Kademlia algorithm calculates the distance between two identifiers using XOR methods. The communication between peers is performed using UDP traffic. Each UDP packet consists of a tuple of $< IPaddress, UDPport, nodeID >$. For more details about this method, please refer to [103]. In table 2.1, we can see that botmasters use Peer-to-Peer protocols to implement different bots. While some of these bots use existing Peer-to-Peer protocols, others have developed custom protocols. In future, we expect more peer-to-peer bots to appear with advance and complex communication features and functionalities.

2.4.8 Examples of Bots

During the past few years, thousands of different bots have been developed. The most notable bots include Eggdrop bots, GT bots, Sdbots, Agobots, Spybots, Phatbots and other Peer-to-Peer bots. A brief description of each type of bots is presented below.

Eggdrop Bot

Eggdrop bot is one of the most well-known and widely used IRC bot. It was written in 1993 by Robby Pointer to monitor a single channel [159]. Eggdrop bot was written in C language. The user can add functionality to the bot since it allows execution of user added Tool Command Language (TCL) scripts. The Eggdrop bot allowed secure assignment of privileges between bots, and sharing the user/ban lists to control floods. These features of Eggdrop bot allowed IRC operators to link many bots together, thus, making it a powerful weapon [159].

Global Threat Bots - GTBots

GTBots started to appear in late 2000. GTBot [69] masters used tools such as HideWindow with their bots to hide the presence of their bots on the infected machines. Some GTBots attempted to spread on to a local network with the help of PsExec. Others made the use of FireDaemon, a service on Windows NT system, to install and run an executable, and IrOffer to act as a file server. GTBots are sometimes hooked on one of the system startup files or launched by a service. Once they start to work, they join the bots master IRC channel. The master starts to issue the commands and the bots respond to these commands. An example of GTBots is Backdoor IRC.Aladinz [159]. Most of the GTBots were used to target individual users, but they could be also used to attack other IRC networks. The GTBots master can order his/her bots to flood the channels with a huge amount of garbage text, which consumes the server bandwidth and as a result, crashes the server.

SdBots

At the start of 2002, a new significant bot development was made by introducing the Sdbot. Sdbot was a well-distributed bot and written in C++, which makes it a very powerful weapon for attackers. In addition, Sdbot has attached its own IRC client within its executable. The early version of Sdbot installs itself onto a registry key so that it can be loaded on startup. Sdbot has added an extra feature to existing bots by using an efficient port tunneling, silent file download and execution, and finally, a flexible C++ source base language [159][12][162][69].

AgoBots

Agobot has been launched in late 2002. Agobot incorporated most features and functionalities of Sdbot, but performed in a more sophisticated and robust manner than

Sdbot. In comparison to Sdbot, Agobot uses other capabilities such as exploits for network propagation, encrypting connections, and polymorphism. As more versions of Agobot were launched, the authors of Sdbot decided to upgrade their bot functionalities as well. In October 2003, a new version of Agobot was released. The new version of Agobot source included new functionalities such as scanners for DCOM RPC, Locator, Webdav service exploits, and a weak NetBIOS password scanner, which makes it function similar to a worm but it requires a botmaster command and does not run automatically [159][12][162][69]. In addition, extra functionalities have been added as Agobot grew. New functionalities include scanning for common backdoors left open by other worms and Trojans, MSSQL server user authentication remote buffer overflow vulnerability, UPnP NOTIFY buffer overflow vulnerability, and the Microsoft Windows workstation service remote buffer overflow vulnerabilities.

Spybots

Spybot is written in C and affects windows system and has similar functionalities and features to the Sdbot. Spybot functionalities include local file manipulation, keylogging, process/system manipulation and remote command execution. In addition, Spybot has the ability to perform different types of scanning such as horizontal scan (single port across range of IP addresses) and vertical scan (single IP address across range of ports), attacking NetBIOS open shares and establishing DDoS attacks such as UDP/TCP/ICMP SYN floods [159][12][162].

The previous bots targeted Windows operating system (OS). Other bots were implemented to target other operating systems. For example, Q8Bot and Kaiten are written for Unix/Linux OS. Like Windows bots, they implement all common bot features such as DDoS attack and executing arbitrary commands. Perl-based bot is also used on Unix-based system. Perl-based bot is simple to implement and contains few hundred lines of source code and mainly used to perform DDoS attack [69].

The first MAC OS X bot is described in [11]. The bot discovered by Mario Ballano Barcena and Alfredo Pesoli has two variants. The bot is used to perform DDoS attack against the website titled dollarcardmarketing.com. According to the authors, the collection of these bots from multiple hosts created the first botnet on the OS X platform.

Peer-to-Peer bots

Many peer-to-peer bots have been implemented in the past. For example, Phatbot uses most of the functionalities implemented in Agobot but rather communicates using WASTE technology instead of IRC protocols. WASTE [172] is an encrypted open-source peer-to-peer protocol for small networks. A detailed description about WASTE protocol, its functionality, architecture, implementation and limitations can be found on [173]. The author of Phatbot removes the encryption from the WASTE code to delete the process of sharing of public keys. In order to find other peers, Phatbot uses a Gnutella cache server [87]. The Phatbot registers itself with a list of URLs using GNUT, a Gnutella client. In order to communicate with other infected hosts, each host connects to these cache servers to retrieve the list of Gnutella clients, but on a different port. In order to connect to the Phatbot WASTE network, a WASTE client is needed and connects to a peer found on the cache servers. One problem when using WASTE client is that the WASTE protocol is designed for small networks, therefore, the scalability issue is raised [131][143].

Other peer-to-peer bots include Sinit and Nugache. The Sinit bot finds other infected hosts by sending special discovery packets on port 53 using random IP address. Once the infected host receives this message, a connection between the two hosts is established and the bots starts to exchange their peers [144][131][171]. According to Florio et al. [37], the Nugache malicious bot is the first bot to build a malicious P2P network. The Nugache bot has a hard coded list of other peers to connect to, on port 8 and limited to 22 servers. The communication between peers is encrypted [131][152].

One of the most dangerous peer-to-peer bots is the Storm Bot (Peacomm). Peacomm uses a well-known peer-to-peer protocol called Overnet [160] on different ports in order to communicate with other peers [37][152]. The infected host sends and receives large number of UDP packets. The bot creates a list of other peers to communicate with it. Some of these peers are legitimate hosts which use Overnet clients.

Table 2.2 shows the summary of the described bots.

2.4.9 Future of Bots

One of the characteristics of the botnet is that it is remote controlled by different mechanisms [69]. The most common command and control structure is the IRC protocol. Other botmasters use different protocols such as HTTP to control bots. We have already seen that some of the existing bots use Peer-to-Peer protocols to communicate and avoid the centralised structure. We expect that most of the future bots will use existing Peer-to-Peer protocols or implement their own Peer-to-Peer networks for communication in their code. A more complicated method will combine different ways of communication. Another characteristic of the bots is the ability to perform malicious tasks [69]. Future bots will not only have more advance techniques to attack other systems but may also use complex techniques such as polymorphic process to hide and evade anti-virus softwares or intrusion detection systems. Yet another characteristic of bots is the way of spreading and propagation to others. As more vulnerabilities start to appear, the bots will try to use these vulnerabilities and find new ones in order to propagate through other systems. Nowadays, most of the bots target windows users. In future, we expect to see bots which targets both windows and (*nix) users [130].

2.5 Previous Botnets/Bots Detection

There are many existing techniques for bots/botnets detection. In this section, we classify these techniques based on types of intrusion detection used (i.e signature-based or anomaly-based) and we will describe these techniques and state the advantages and disadvantages of these techniques. We will also show how our system of bots/botnets detection is different than the existing techniques.

Table 2.2: Examples of existing IRC and P2P bots.

Bot's Name	Protocol	Description
Eggdrop Bots	IRC	first non malicious bot written in C, allows the user to add functionalities
PrettyPark	IRC	first malicious bot uses command and control
Subseven	IRC	trojan bot
GT Bots	IRC	malicious bot uses IRC scripts, targeted individual user, spread to local network, written in C++,
SdBots	IRC	uses port tunneling, file download and execution
AgoBots	IRC	similar to Sdbot's functionalities with additional propagation and encryption methods
SpyBots	IRC	similar to Sdbot's functionalities with file and process manipulation and keylogging
rxBot	IRC	descendant of sdbot
PhatBots	P2P	same functionalities as Agobot but uses WASTE protocol
Bobax	HTTP	bot uses http protocol for command and
ClickBot	HTTP	same as Bobax, used for click fraud
SpamThru	P2P	bot using custom protocol for backup
Sinit	P2P	find other peers by sending a discovery message
Nugache	P2P	first bot to build malicious P2P network, implemented its hard coded peerlist in order to other peers
Peacomm	P2P	creates a list of other peers to communicate with using UDP packets

2.5.1 Botnet Taxonomy

Govil [45] presents a general description of botnet, what are they, their life cycle, communication protocols, malicious activities and highlights various detection mechanisms and how to defend against these bots.

Barford et al. [12] analyse the source code of the most popular existing bots and classify them based on (1) host control mechanism by hiding bots from being detected, manipulating files and harvesting local information (2) propagation mechanism through scanning techniques (3) exploit mechanism through attacking known vulnerabilities (4) malware delivery mechanism and (5) obfuscation and deception mechanism by hiding the bot's transmitted data on the network to evade the detection applications such as anti-viruses and network intrusion detection systems. A more technical description of how these bots operate is presented by Thing et al. [162].

Dagon et al. [30] describe different botnet topologies based on their utility of the communication structure and their corresponding metrics, which measure the botnet's effectiveness, efficiency and robustness and provide response techniques to degrade and disrupt botnets. They define the effectiveness of the botnet as an estimation of overall utility such as spam, DDoS, wares distribution or phishing to achieve a given purpose. They also measure the efficiency of the botnet which includes forwarding command and control messages, updating bot code and retrieving host-based information. Finally, they measure the robustness of the botnet to estimate the degree of connection between bots which perform sensitive tasks such as spamming or storing files for downloads.

Trend Micro [165] presents some existing and new classifications of botnet structures. These classifications are based on attacking behaviour, command and control models, rallying mechanism (mechanism for new bots to locate and join existing botnets), communication protocols, botnet activities and evasion techniques. The classification which is based on attacking behaviours includes DDoS attack, scanning, remote exploitation, spamming, phishing, spywares, and identity thefts. Command and control classification include classifying botnets based on centralised, distributed, peer-to-peer, and randomized structures. The rallying mechanism classifies botnets based on the ability of discovering other bots which include hard-coded IP, dynamic DNS and distributed DNS services. Botnets can also be classified by their communication protocols such as IRC, HTTP, Instant Messages and Peer-to-Peer protocols. In addition, they can be classified according to their activities such as DNS queries, burst packets and system calls. Finally, botnets can be classified according to their evasion techniques which include packing, tunneling command and control communications and encrypted commands traffic.

Abu Rajab et al. [127] explore different mechanisms to determine botnet sizes through botnet infiltration by joining the command and control channel to examine bots malicious activities, DNS redirection and cache probing and finally queries made to DNS blacklists. They show that the size of the botnet varies based on the way the bots are counted. For example, nicknames identification counts show that there are more than 450,000 bots while counting the IP addresses yields sizes in the range of 100,000 bots including the active ones. They conclude that the number of active bots is normally constrained by the capacity of botnet server and affected by high bot churn rates. They also state that there are several implications which affect the task of estimating the actual botnet members. This include when the bots migrate from one botnet to another, when the botmaster creates the bots clones and when the bots join different command and control channels on the same server or connect to a different server together. They also discuss the hidden relationship between different botnets to estimate the actual bots size by removing the overlaps.

In another paper, Abu Rajab et al. [128] collect bots and track botnets by using botnet infiltration to identify bots which participate in botnet network. They have developed an IRC tracker which mimics the behaviour of actual bots and records all the information observed on the command and control channel and use this information to identify active bots. Their approach is based on the fact that bots normally make a DNS query to resolve the IP address of their IRC server before joining the channel. By probing and recording the DNS cache hits, they have estimated the total number of the infected hosts. The cache hit means that at least one bot has queried the IRC server within the time-to-live interval of the DNS entry corresponding to the botnet server.

Ramachandran et al. [129] locate spam bots by implementing a DNS blacklist (DNSBL). In their approach, they assume that the botmasters may use the DNS black list to know the bots status. Their approach may generate many false alarms. As a result, this approach is most useful for detecting spam botnets and does not show which botnet a specific bot belongs to.

Dagon et al. [31] develop a technique for counting infected hosts by changing the DNS entry for a botnet's IRC server and redirect the connections to a local sinkhole. The sinkhole completes a 3-way TCP handshaking to establish a connection with bots attempting to connect to the redirected IRC server and record the IP addresses of these infected hosts. They found that large botnets may contain more than 350,000 bots. Their technique based on counting the total number of infected bots including inactive bots based on the connection attempts. In addition, the sinkhole does not run an IRC server which means that it is difficult to know if these bots are connecting to the same command and control channel.

2.5.2 Honeypots

The use of honeypots allows the researchers to examine bots' binaries, monitor their behaviours and then extract signatures to detect them in the future. A Honeypot is an entire network or server that is not meant to provide any service to legitimate users [179][3]. The only traffic that a honeypot receives is probe traffic or other malicious traffic. Thus, any traffic that is captured by the honeypot is assumed to be unauthorized and malicious, and can be analysed to identify any threatening activities. Attack signatures can then be obtained from captured traffic, and used to block attacks into real systems [100]. The disadvantage of a honeypot is that it cannot detect the suspicious traffic without receiving activity directed against it. Therefore, the need to monitor connections between hosts is required. Furthermore, attackers can potentially identify a honeypot based on certain expected characteristics or behaviours. For example, a honeypot can emulate a Web server to send an error message to the attacker [141]. If there is any unique formatting in the message, the attacker will notice the difference, and thus try to attack and control the honeypot in order to harm other systems. Moreover, honeypots are designed to capture malwares which use scanning for vulnerabilities technique to propagate but they cannot detect malwares that use different methods to propagate (e.g. emails). Another limitation is that current malwares are designed to determine if virtual machines are used as honeypots [178][70].

Tracking Botnets using Honeypot

Honeypots can be used to track and observe botnets [9]. Honeypots have special software to collect data about the system behaviour. By default, a honeypot is a resource that is exploited by automated malware such as a bot. Once the bot is installed in a honeypot, it tries to connect to the IRC server and joins the required channel to obtain further commands. The outbound connections from the honeypot give a clear indication of the infection.

Freiling et al. [40] try to prevent DDoS caused by the botnet by exploring a root-cause methodology. Freiling and his group use a non productive resource (honeypot) by setting up a vulnerable system to be infected with a bot. Once the honeypot is infected, they collect the required information to infiltrate the botnet network. Their approach of detecting the botnet is based on penetrating the remote control network by using an agent, analyzing the network traffic to gather information about the botnet and finally, using the collected information to shut down the remote control network.

The detail description of their approach is as follows: Freiling et al. have used a Honeywall bridge that enables the Data Control and Data Capture tasks. The need of controlling and gathering as much information as possible passing between a client (a bot) and an IRC server is an essential task. Controlling data allows controlling outgoing connections that pass between a bot and a botmaster. In addition, gathering data (IP/DNS address and port number) can be used by some kind of data capture programs (e.g. snort [138] or tcpdump [158]). Once all the required information is obtained, the collected data can be used to join the channel and track other botnets in different networks. Another method of collecting information is by reverse engineering and then analysing the captured data to extract required information. Reverse engineering can be time consuming and difficult to apply when the captured data is encrypted. The previous approach has many disadvantages such as:

- A honeypot cannot detect traffic unless it is directed to it. Therefore, if a bot fails to exploit the offered services provided by the honeypot, the honeypot will be useless.
- The honeypot needs to be monitored to detect abnormal behaviour in the system.
- Honeypot cannot handle large numbers of IP addresses because it has small number of resources and collects small datasets of attack connections.

Due to these reasons, a program called mwcollect is introduced. Mwcollect is an easy solution to collect malicious traffic in a non-native environment like FreeBSD, and Linux. Mwcollect simulates several vulnerable services and waits for them to be exploited. In contrast to honeyd [125], it is tailored to the collecting of malware and provides better packet handling. Mwcollect has a core module to handle network interfaces, coordinate the actions of other modules, and implement a sniffer mode to record all traffic. There are four types of modules that are registered in the core. These modules include vulnerability modules which open common vulnerable ports, shellcode modules which analyse shellcode received by one of the vulnerability modules, fetch modules which download the files, and submission modules which handle successfully downloaded files. There are two important features of mwcollect that makes it work efficiently: virtualized file system and shell emulation. Virtualized file system allows infecting hosts via a shell by writing commands for downloading and executing malware into a temporary file and then execute this file. Shell emulation emulates windows shell to allow malware to spread if it did not spread by downloading shell code. The main advantage of mwcollect is to provide stability and scalability. This is because mwcollect can be expolited by all types of attacks as compared to honeypot (running Windows 2000) which will be rebooted if the bot tries to exploit it with payload that targets Windows XP. In addition, mwcollect can listen to many IP addresses in parallel.

Cooke et al. [26] illustrate the botnet phenomena and the effectiveness of detecting botnets. In order to detect the botnet, they have placed a vulnerable host which acts as a honeypot behind a transparent proxy device. The proxy is used to limit the rate of traffic, disallow access to the local network, and log all traffic from/to the honeypot. Once the bot infects the honeypot, the command and control characteristics of outgoing traffic connections are identified by locating all successful outgoing TCP connections and verify that all connections belong to command and control activity through inspecting the payloads.

Three approaches for handling botnets are investigated. The first approach is to prevent the system from being infected. They discover that preventing all systems on the Internet from being infected is an impossible task. The second approach is to directly detect botnet command and control communication between bots, and between bots and controllers. As most of the bots are controlled using IRC, the second approach is achieved through: 1) monitoring standard ports used for IRC traffic (port 6667), 2) inspecting payload for strings that match known botnet commands, and 3) looking for the behaviour characteristics of bots that differ from non-human characteristics. They found that botnets can run on non-standard ports, inspecting payload packets is very costly on high throughput networks, and there are no simple characteristics of communication channels that can be used for detection. The last approach for handling botnets is to detect botnets by identifying secondary features of a bot infection such as propagation or attack behaviour by using the correlation of data from different sources to locate bots. The main idea is to track botnet by monitoring the bots propagation activities. Most of the bots are configured to propagate by exploiting vulnerabilities. Therefore, monitoring the number of packets received to a specific port usually gives a good indication of malicious activity.

2.5.3 Signature-based Detection

Stephane Racine [126] summarizes three steps for a possible botnet detection algorithm. The first step is to find inactive clients. This is done by recording the IRC PONG messages and assigning them to a connection which will help in detecting inactive clients. The second step is to classify inactive clients by channel membership. As a result, a large group of inactive clients belonging to the same channel would be suspicious. The final step is to analyse IRC traffic by channel and search for characteristics of botnet traffic. This approach was successful in detecting idle IRC activity but suffered from high false positive rates.

Bolliger and Kaufmann [19] further develop the ideas mentioned by Racine and used three approaches to detect bots and botnets. The first approach is to consider the fast joining bots. These bots are installed by a propagating worm and they will join the IRC network in a short period of time. Thus, by detecting a fast increase in the number of IRC clients on an IRC server gives a good indication of fast joining bots. The second approach is to detect the long standing connections to an IRC server which represents idle bots waiting for master commands. Some properties were taken to detect these type of bots such as duration of connection, amount of data sent from/to IRC server/client, and the number of IRC PING-PONG. The last approach was to monitor IRC connections that mostly consist of PING-PONG traffic and no real conversation traffic.

Strayer et al. [153] try to detect the botnet by examining flow characteristics (network traffic). Their architecture is based on filtering network flows that are unlikely to be part of the botnet. Five filters are used to achieve this task. These filters include TCP-based flows, a complete TCP 3-way handshaking, no high-bit-rate flows, no flows with a packet size greater than 300 bytes and finally no flows with short duration. After the filtering stage, the remaining flows are classified by using a machine learning algorithm based on chat-like characteristics such as timing and packet size characteristics using different machine learning classification algorithms. The third stage is to keep the flows that are active for a long period of time. The forth stage is to use the correlator to examine the pair-wise flows that have common similarities which are part of the same botnet. The final stage is to pass the remaining flows to the topology analyser to determine which flows share a common controller. Such an approach can be easily defeated by changing network flows. In addition, their approach did not consider the correlation of events generated by bots which enhance the bot detection method and reduce false alarms.

Goebel and Holz [44] implement a *Rishi* which is a signature-based approach that detects bots by monitoring network traffic looking for suspicious IRC nicknames, IRC servers and uncommon IRC ports by detecting the communication channel between the bot and the botnet controller. This is done by extracting network packets containing IRC related information such as nickname and password and then analysing suspicious hosts by passing the information to a function to calculate the scoring value. This function looks for the occurrence of suspicious substrings, special characters or long numbers which increase the value of the scoring function. If the value of the scoring function exceeds some threshold, the host is likely to be infected with the bot. Because their approach is based on using bot's nicknames as signatures, Rishi needs to know all available signatures to detect all types of bots. In addition, this approach can be evaded by using different methods of generating nicknames. Moreover, if the communication between botmaster and bots are encrypted, this approach might not work well.

Gu et al. implement different intrusion detection applications such as BotHunter, BotSniffer and BotMiner. BotHunter [60] is a real time network-based botnet detection system. The BotHunter consists of the correlation engine which detects specific stages in sequences during the malware infection process. The Bothunter receives events/alarms from different sources and correlate the inbound intrusion alarms with the outbound communication patterns to indicate that a host is infected with a bot. A combination of five correlated events is used. These combinations are inbound port scan, inbound exploit, internal-to-external bot binary download and execution, internal-to-external command and control communications and outbound port scan. The BotHunter uses different anomaly detection techniques to detect these events such as Statistical sCan Anomaly Detection Engine (SCADE) which is used for port scan analysis for incoming and outgoing network traffic, Statistical payload Anomaly

Detection Engine (SLADE) which implements a n-gram payload analysis and rule-based snort signatures detection engine. The BotHunter declares that the bot is infected when there is an incoming infection warning followed by outbound local host coordination or exploit propagation warnings or a minimum of at least two forms of outbound bot warnings such as bot binary download, C&C communications and outbound scan.

2.5.4 Anomaly Detection

Using an anomaly-based intrusion detection method, Gu et al. implement a BotSniffer [61] which is a network-based anomaly detection that identify botnet command and control channels without prior knowledge of signatures. BotSniffer detects bots by examining the correlation and similarity patterns between bots activities within similar time window such as coordinated communication, propagation, attack and fraudulent activities due to the pre-programmed response activities to botmaster commands. They hypothesize that in the normal network service, it is unlikely that many clients respond at a similar time. The BotSniffer consists of two engines, the monitor engine and the correlation engine. The monitor engine filters out irrelevant traffic such as ICMP, UDP traffic and well-known traffic to reduce the traffic volume, records any suspicious C&C protocols and finally detects message response behaviour by monitoring IRC PRIVMSG messages for further correlation analysis and activity response behaviour by using SCADE described earlier. Once these events are collected they are analysed by the correlation engine which uses the clustering method to find similar activity or message behaviours.

BotMiner [59] is a network-based botnet detection that is independent of command and control protocol and structure. BotMiner works by clustering (1) hosts with similar communication traffic that identifies which hosts are communicating with other hosts and, (2) hosts with similar malicious traffic which identifies activities on that host. The communication traffic includes some statistical information such as flow per hour, packets per flow, bytes per packet and bytes per second while the malicious traffic include malicious activities such as scanning, sending spam and downloading executable binaries. Then they apply the clustering algorithm to classify groups based on communication patterns and activities pattern and perform cross cluster correlation to identify the hosts with common communication and malicious activity patterns.

Karasaridis et al. [86] develop an anomaly-based algorithm which detects IRC botnet controllers running on random ports without the need for pre-defined signatures by using the transport layer data flow. To detect the botnet first they identify hosts with suspicious behaviour such as scanning, spamming or participating on DDoS attack. In addition, they record the duration of these malicious activities and the links where such activities are detected. The next step is to fetch all data flows from/to these hosts and process the flows that represent connections to controllers (referred to as candidate control flows). Second, they analyse the flow activity to identify candidate control flows and store them in conversations which summarize the flow records between local hosts and remote hosts on particular remote ports. The candidate control flows for each local host IP, remote host IP and remote port are summarized in a candidate control conversation which is a conversation between a suspected bot and a remote host that satisfies certain conditions. Two approaches are used to identify the candidate controller connections to uncommon IRC ports. The first approach is to identify flow records between the suspected bots and remote hosts that appear to be hub servers. The second approach is to find records whose characteristics are within the bounds of a flow model for IRC traffic. Third, they analyse the candidate control conversation to isolate the suspected controllers and controller ports. The analysis include 1) calculating the number of unique suspected bots for given remote server address/port with consideration of the most known servers, 2) calculating the distances between the traffic to remote server ports and the model traffic by giving equal weight to each flow characteristic with the consideration of those servers who are below threshold and 3)calculating a heuristics score for server address/port pairs that remain candidates based on the number of idle clients and whether the servers uses TCP/UDP on the suspected ports or has many clients. Finally, reports and alarms are generated based on the analysis. Their analysis is based on detecting IRC bots by matching IRC traffic. While this approach is suitable for detecting IRC bots, it may not be suitable for detecting other types of bot such as P2P bots. In addition, their approach is based on flows characteristics which can be defeated by changing network traffic.

Binkley and Singh [16] develop an anomaly-based algorithm for detecting an IRC-based botnet mesh which combines the IRC mesh components with a syn-scanner detection system. The IRC mesh components contain statistics about TCP packets and define IRC channels as a set of IP hosts. Then they correlate these IP hosts over a large set of data sampled to identify which host in the IP channel was a scanner. They also define a TCP work weight metric for every host as the ratio of TCP control packets sent to total TCP packets sent which classify if a particular host is participating in DDoS attacks. The algorithm works for certain types of botnets which perform scanning activity in order to detect botnet.

All previous methods use network-based intrusion detection systems. Stinson and Mitchell [151] develop a host-based anomaly detection, BotSwat, to detect bot behaviour on the system by monitoring execution of arbitrary Win32 library. The Botswat is based on Detours [75] to monitor API function calls. Detours is a library for dynamically intercepting API function calls. Detours works by replacing the first 5 bytes of target function with an unconditional jump to the monitored function provided by the user. This interception program is useful because it works in run time and can intercept function calls whether it they are statically linked, dynamically linked or delayed in loading to memory. Their idea is based on the fact that the botmaster sends the commands to the bots over the command and the control structure and the bot responds to these commands by invoking system calls. They try to distinguish remote execution of bot's commands from local execution of commands of benign programs by identifying the arguments of system calls. These arguments contain the data received over network which may be used in malicious bot activities. Three components are used to perform this task, tainting component, user input component and behaviour-check component. The tainting component is to identify any process that receives untrusted information over the network which may contain the botmaster commands. The data received is considered tainted and tracked as it propagates via library calls to other memory regions. To identify botmaster commands, the tainted data is supplied to the selected specific gate functions which are system calls used in malicious bot activity. When a process uses tainted data in a system call argument, Botswat identifies the execution of the command matching and correlating the system calls invoked and the received network data in the system call arguments with system calls and arguments in the function gates.

Stinson also introduces some techniques to reduce the effect of out-of-band memory copies such as hiding the path of tracked data. The first technique is to use a content-based tainting which classifies the memory as a tainted memory region if it contains identical string to a known tainted string. The second technique is to use a substring-based tainting where the region is tainted if it contains a substring of any data received by the monitoring process over the network. By applying these techniques, they claim that it is easy to detect bot behaviour even when all of the bot's calls to memory copying functions occur out-of-band. The user input component is to identify actions or data values that are initiated by local applications which correspond to mouse or keyboard input events. Finally, the behaviour-check component queries the tainting and user-input components to determine whether to flag the invocation of selected system calls as exhibiting external control. This approach may suffer from a large number of false alarms. First of all the detection of bot behavior is based on the selection of proper API function calls and their arguments. In addition this approach may generate false positives because some legitimate programs may use a portion of network traffic in some of their API function call arguments. Furthermore, processing API function calls with their arguments using pattern-matching technique can increase the analysis stage and reduce the performance. Because this approach is based on evaluating the arguments of system calls, it can be easily defeated by encrypting or manipulating the received data over the network. Thus, the comparison between the arguments of the received data over the network and arguments in the functions gate will be difficult unless the received data over the network is decrypted.

Another host-based malware detection introduced by Cui et al. [28]. BINDER is an extrusion-based break in detector which correlates stealthy outgoing network connections and processes information with user input, based on the assumption that malicious software runs in the background and generates connections without user input. The collected information is passed to an extrusion detection which detects malicious processes based on the collected information. One of the problems of BINDER is that it flags the processes as malicious if they are created without user input. Thus, if the user executes a malicious process, this process might not be detected.

2.5.5 Machine Learning

Machine learning techniques are also used in detecting bots. Machine learning algorithms do not need explicit signatures to classify malware programs but rather is based on finding common features and correlating different activities of the malware. For example, Strayer [153] and Livadas et al. [99] propose machine learning techniques for botnet detection by using network statistics. These statistics include bytes per second, duration and packet per second of some protocols that are used for chatting such as IRC protocol. Their methods contain lots of heuristics rules in order to reduce the flows. In their book, Strayer et al. [154] publish a detection approach to examine network flow characteristics such as bandwidth, packet timing, and burst duration to find an evidence of botnet activity by first filtering unrelated traffic to botnet to reduce the amount of data being processed. The second step is to classify the remaining traffic into a group that is likely to be part of a botnet using three machine learning classification algorithms, namely J48 decision trees, Naive Bayes and Bayesian Network. The final step is to correlate the remaining traffic with each other to find clusters of flows that share similar timing and packet size characteristics that is part of the activity of a botnet.

Another paper by Nivargi et al. [115] uses a different kind of machine learning classification algorithm such as multinomial Naive Bayes, linear SVM, kNN and J48 decision tree to classify binaries into malicious or benign programs. The features that they have used are based on the number of users that the IRC channel has, means and variance of words per line of non-human words, number of frequency of IRC commands, and number of commands that are used.

Kondo et al. [94] used machine learning technique support vector machine (SVM) in order to classify the behaviour of botnet by monitoring the C&C session. They claim that the SVM has better classification functionality, accuracy and can process high dimensional vector data. In their method, they have shown that they can distinguish botnet behaviour from the human behavior when using different direction of C&C. They define three vectors for the C&C session classification: session information vector, packet sequence vector and packet histogram vector. The packet histogram vector is used to transform network traffic to high dimension. The result showed that the packet histogram vector achieves a high detection accuracy in comparison to other two vectors. They also have compared their method with other machine techniques such as Naive Bayes and k-Nearest Neighbor (k-NN) classification methods. They claim that the SVM generates better results, has a better classification accuracy and faster processing speed than the other algorithms for classifying the session information. The Naive Byes misclassified all sessions as the C&C session in all feature vector data format. In contrast, the training processing cost for SVM was higher than the other two methods but they claim that the training for SVM is performed only once at the beginning. Their method is just applied to IRC bots and no other experiments are performed for non-IRC protocol.

Tracking Peer-to-Peer Bots

A number of research works have been conducted to analyse and detect peer-to-peer bots. For example, Schoof and Koning [131] analyse different peer-to-peer bots such as Sinit and Nugache to examine the behaviour of these bots. In their analysis, they note that some peer-to-peer bots communicate on a fixed port. They argue that by monitoring traffic on that port, one could detect these bots. They also discover that some of these bots generate a large number of destination unreachable error messages (DU) and connection reset error messages while trying to connect to other peers. In addition, some bot's communications are encrypted, which make the traffic analysis a difficult task and results in high false alarms.

Dittrich and Dietrich [33] explain some of the features and challenges when dealing with the Nugache P2P botnet. Stover et al. [152] conclude that there is no static IDS that will detect Nugache traffic. They also mention that the Nugache bot can be detected through various signatures of the infection.

Holz et al. [71] present a method to analyse and mitigate P2P botnet. First, they exploit the peer-to-peer bootstrapping process to stimulate the infection process and extract the connection information. By retrieving this information, they are able to join the botnet network and receive commands. They develop their crawler which runs on a single machine and uses a breadth first search issuing route requests to find the peers participating on peer-to-peer networks. Finally, because they consider unauthenticated communication, they argue that they can inject commands into the botnet and disrupt the communication channel. They also develop ways to mitigate Storm worm and introduce an active measurement technique to enumerate the number of infected hosts. Their approach is based on either reverse engineered the bot binary to identify the function which generates the key that is used for searching for other infected machines and bots commands or use honeypot and infect it with the bot that generates a new key each time it is rebooted and thus enumerate all the keys. The problem with the first method is that the process of reverse engineering is needed every time the attacker changes the key generation functions. The other method takes long time to enumerate all the keys.

Other researchers analyse different peer-to-peer bots such as Storm bot (Peacomm) where large number of emails are spammed to many accounts holding an executable attachment [37]. An in-depth analysis of Peacomm is provided by Stewart from SecureWorks [145]. Nunnery and Kang [116] try to locate the zombie nodes activities in peer-to-peer network by their retrieval of hashes and the control of a large group of network computers. They claimed that if the client within the controlled network searches for hash used by malware, it must be a zombie node. This process can also leads to locate the IP address of the botmaster by monitoring any publish activity on the supervised network.

A detailed description of Peacomm is presented by Porras et al. [123]. They also investigate how to detect the Storm bot by using a BotHunter which tracks the two-way communication flows between internal and external entities to find the infected host. Stover [152] suggests that the Storm bot can be detected by configuring IDS to find the configuration file used by the bot. They also state that differentiating between the Storm bot and legitimate P2P communications is a difficult task.

Stewart [146] state that the Peacomm died on September 18th, 2008 because of some mistakes in the encryption protocols, some discover ways to disrupt the botnet.

2.6 Critical Assessment and Relation to our Work

Many existing botnet/bots detection techniques which are discussed previously have advantages and disadvantages over others. In this section, we will state the limitations of the existing techniques and suggest solutions which can improve their performance.

Different botnet detection techniques can be classified based on different criteria such as either they are a network-based intrusion detection or a host-based bot detection, detect a group of bots (botnet) or an individual bot, a signature-based bot detection or anomaly-based bot detection, restricted to one type of command and control structure or can detect bots that uses multiple command and control structure. We already stated the disadvantages of using honeypots to track botnet by infiltrating the command and control channel. As a result, we exclude this option from our implementation and focus on developing a new monitoring technique to track and collect bots traffic.

Many existing botnet/bots detection methods are network-based (e.g. BotMiner [59]). The disadvantages of network-based detection were already discussed in section 2.3.1. Other techniques use a host-based detection to detect the botnet such as BotSwat [151]. BotSwat is introduced as the only host-based bot detection method to monitor the behaviour of programs by monitoring the network data received as arguments of the selected system calls. This method can suffer from false alarms as many benign programs could receive network data in their system calls arguments. In our research, we notice that most of the existing techniques that are implemented use network-based bot detection, and there are few techniques for a host-based intrusion detection. As a result, our choice was to develop a host-based intrusion detection method which can detect malicious programs in our system.

After deciding to use a host-based bots detection, our concern is either to detect a group of bots (botnet) or an individual bot. Since most of the existing research on botnet detection focuses on botnet [99][86][61][59] rather than an individual bot, we started our detection technique by implementing a botnet detection algorithm based on log correlation. The basic idea is that different bots respond to botmaster commands by exhibiting similar activities simultaneously. We record these activities by monitoring system calls generated by these bots and save them in a log file. After that, our algorithm checks for the change of rate of the sizes of the log files and finds any existing correlation which indicates botnet activities. Part of this idea was implemented by Gu et al. [61][59]. BotSniffer is used to detect the bot by finding the correlation of similar patterns in network traffic which are caused by similar activities while BotMiner [59] identifies hosts with similar communications activity and similar attack traffic. From this experiment of botnet detection, we have noticed that botnet detection could be easily detected by correlating different activities from different resources as suggested by Cooke et al. [26]. As a result, our main task is to implement a host-based bot detection that detects an individual bot. Therefore, we have used the same idea which correlates different bot activities within a specified time window in one host. Note that BotMiner uses a statistical approach to correlate events in order to detect botnets while our approach monitors events which generates similar system calls to detect botnets.

The third issue is either to implement a signature-based bot detection or anomaly-based detection. Most of the existing techniques for bots detection use signature-based approach to detect bots. For example, Rishi [44] is signature-based detection which monitors network traffic looking for suspicious IRC nicknames, servers and uncommon ports. Although signature-based bot detection techniques are an excellent way of detecting previous bots which already have signatures, they cannot detect new bots. In addition, previous bots can update their signatures to evade signature-based bot detection technique. Signature-based bot detection might fail if the communication between the botmaster and the bots are encrypted because they usually examine the content of received packets. Moreover, since the bot's environment is based on scanning, flooding, spamming or stealing sensitive information from the user host, it is more appropriate to use anomaly bot detection. Therefore, our focus is to implement an anomaly/behaviour bot detection. Although there are some existing anomaly/behaviour bots detection described previously, none of them monitor bots behaviour based on their system call execution except for the method implemented by Stinson [151]. Although Stinson et al. uses a host-based bot detection, their approach based on monitoring the arguments of system calls which can be defeated by encrypting the commands. Our approach does not examine system calls arguments but rather focuses on some selected system calls which can be used by bots to perform malicious activities. In addition, none of the existing techniques use an artificial immune system which correlates different signals to detect the existence of individual bots on the host. As an example, BotHunter [60] correlates different events within a fixed time window from inbound scan to outbound scan to detect the presence of bot. The BotHunter relies on detecting bot at the network level without considering local attack such as keylogging activities and deleting files. In our case, these events are taken into account and by using the artificial immune system approach we are able to correlate different events within a flexible time window.

The last issue is either to restrict our method to one type of command and control structure or multiple types. Some of the existing techniques implemented by Goebel [44] and Binkley [16] are IRC-based botnet. Other existing techniques implemented by Livadas [99], Karasaridis [86] and Gu [61] detect bots which use a centralised structure. The problem with these techniques is that they might not detect bots which use different types of command and control structures or use different protocols to communicate such as HTTP or peer-to-peer protocols. Our focus was to implement an algorithm which is suitable for different types of communication and topology and does not depend on a specific type. Table 2.3 summarises some of the existing botnet/bots techniques and their specifications and features.

- SB: Signature-Based Detection with network statistics
- AB: Anomaly-Based Detection with network statistics
- NB: Network-Based Detection

Table 2.3: Existing Techniques Characteristics

Technique	SB	AB	NB	HB	Botnet	I-Bot/AIS	IRC-S	P2P-S
BotHunter		x	x		x		x	x
BotMiner		x	x		x		x	x
BotSniffer		x	x		x		x	
Livadas			x		x		x	
Karasaridis	x		x		x		x	
BotSwat				x		x	x	
Rishi	x		x		x		x	
Al-Hammadi		x	x	x	x	x/x	x	x

- HB: Host-Based Detection
- I-Bot: Individual Bot
- IRC-S: IRC Structure
- P2P-S: Peer to Peer Structure
- AIS: Use Artificial Immune System

Chapter 3

Methodology

3.1 Introduction

Most of the existing detection methods use network analysis to detect botnet/bots. As we explained in the previous chapter, these methods face different problems. One of these problems is that they use signature-based detection to monitor botnet activities. These methods might fail to detect bots in case new bot signatures appear. In addition, if the communication between the bots and the botmaster is encrypted, many existing botnet detection algorithms will not work properly. Furthermore, existing methods focus on detecting botnets rather than individual bots. Moreover, many existing methods focus on detecting one type of botnet topology such as IRC bots.

In order to address these shortcomings of the existing botnet/bot detection methods, we introduce a framework which will detect both, botnets and individual bots running on the system. Our framework uses a different approach from the existing methods. Our approach mainly focuses on detecting an individual bot with a secondary botnet detection algorithm. Rather than using a signature-based detection, our approach is based on monitoring specified API function calls executed by the processes to examine the behaviour of these processes. To the best of our knowledge, none of the previous methods on botnet detection monitor the execution of function calls except the one that has been introduced by Stinson [151]. One of the main differences between our approach and the Stinson method is that we are not considering function calls arguments in our approach. As a result, if the data is encrypted, our approach can still detect the existence of the bot. The second main difference is that we are considering the correlation of different activities of a single process to detect the malicious behaviour. In addition, we use an artificial intelligence algorithm the dendritic cell algorithm - DCA.

Our main contribution is that we focus on monitoring specified API function calls executed by the processes in order to detect malicious activities. Therefore, we developed a tool, APITrace which is implemented with the help of [73], to monitor the most frequent function calls used by bots to perform malicious activities. The reason why we have developed our own monitoring program instead of using existing techniques such as StraceNT [41] or Systrace [124] is that we needed to monitor only important function calls that we think that they can be used for malicious activities instead of monitoring large number of functions calls. This will reduce the processing time by the algorithm. These function calls will be discussed later in this chapter. In addition, we needed to meet the input format of some correlation algorithms that we have used and design our monitoring program accordingly.

We also developed a framework for detecting an individual bot running on the system. This framework works for both detecting an individual bot and botnet by correlating different activities within a specified time window. We also use an intelligent way of correlating these activities to enhance the detection mechanism. We have modified the old anomaly score termed mature context antigen value (MCAV), used in the DCA, and have presented a modified anomaly score termed MCAV Antigen Coefficient (MAC) to increase the detection sensitivity and reduce the false positive alarms generated by the MCAV value. The framework can detect IRC bots in addition to the Peer-to-Peer bots and does not depend on one type of botnet structures.

This section is structured as follows: we describe the Windows system architecture in section 3.2. We discuss different function calls interception techniques and how we developed our monitoring tool in section 3.3. Our framework of botnet/bot detection is explained in section 3.4. An overview of the design and implementation of detecting botnets through log correlation is described in section 3.5. We also discuss the design and implementation of detecting an individual bot by using Spearman's rank correlation (SRC) in section 3.7 and by using DCA algorithm for both IRC bots and P2P bots in sections 3.8 and 3.9 respectively. We summarise and conclude in section 3.10.

3.2 Windows Architecture and Windows API function calls

The Windows operating system consists of two modes, a user mode which is called *Ring 3* and a kernel mode which is called *Ring 0* [161] as shown in Figure 3.1. All window kernels reside on Ring 0 while all other windows applications reside on Ring 3. Applications on Ring 3 cannot interact directly with CPU, instead, they first communicate with the kernel on Ring 0 which then requests the operation to be performed by CPU. In order for applications running on Ring 3 to communicate with kernel on Ring 0, they need to send requests to the kernel, therefore, Windows has developed functions that allow applications on Ring 3 to request operations to be executed by the kernel. These function calls are called Windows Application Programming Interface - API function calls. These function calls make the system stable and allow developers to perform low level operations.

To explain this process, when a program needs to perform some action, it calls specific API function calls by passing their arguments in order to perform different tasks. These function calls in turn call other function calls which resides on kernel mode.

To achieve the task of detecting botnet/bot, our framework mainly depends on monitoring specified API function calls. In order to do so, we have developed a program called *APITrace* to monitor selected API function calls. The APITrace is a tool that provides a way to monitor an application's behaviour by means of intercepting API function calls issued by the monitored processes at a user level. To examine the behaviour of any process, the process generates API function calls in order to achieve some task. The generated API function call is intercepted by our APITrace tool. In this section, we will address different methods of intercepting these function calls and we will describe how these function calls can be helpful for detecting malicious processes.

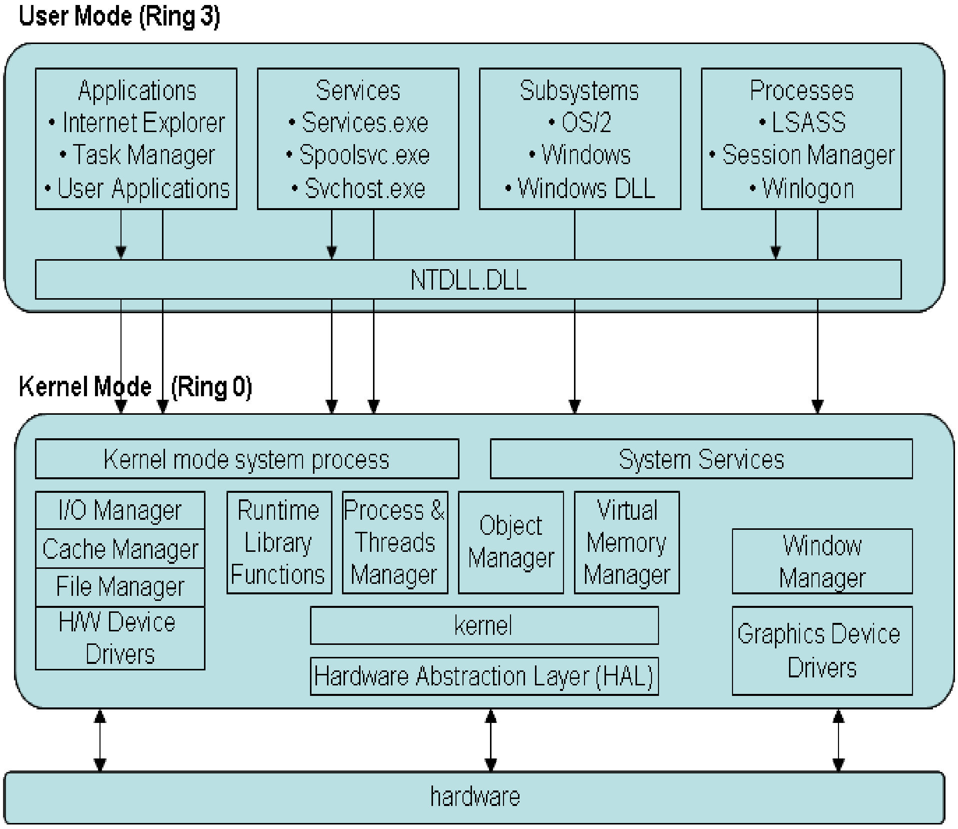

FIGURE 3.1: Windows Architecture showing the User Mode and the Kernel Mode.

3.3 API Hooking

The interception technique [82], known as hooking API function calls, is the process of intercepting function calls to identify the actions taken by the program to perform different tasks. It allows the user to understand how these function calls were invoked and brings a better understanding of how the program behaves. In a Windows environment, many API function calls are not well documented, therefore, by using the interception technique, the software programmers gain more information about these functions and how they operate. In addition, interception techniques allow the programmers to control another process or change the behaviour of this process by extending their functionalities or by adding extra features according to the programmers needs, or by determining how this process will interact with other processes. Monitoring API function calls allows the programmer to control these function with their arguments and track specific actions generated by the process. For example, anti-virus softwares need to map some files into the target process to control its behaviour. In addition, debuggers need to map some files into the target process in order to detect, locate and correct errors generated by these processes. Security check tools need to map files into target process to verify that the user is allowed to execute this process or to run a particular task [42]. This is all performed by monitoring specified API function calls through the hooking process. We have presented some of the methods used to perform API monitoring to allow the reader to get a basic idea behind this process.

There are different types of interception techniques such as system-wide hook, proxy DLL, entry point rewriting and import address table patching. System wide hook, also known as windows message hook, is based on capturing windows messages using callback functions before it reaches the processing part. These functions, lies in the DLL file. When Windows operating system maps the DLL file into the target address space, filter functions are called instead. Proxy DLL hook is an easy way for intercepting API function calls by exporting all API function calls in the original DLL file to the proxy DLL file. In this section, we will present different intercepting techniques which helped us to develop a tool to monitor some of the interesting API function calls used by malicious processes. The last two methods are based on code modification hook by either changing the first few bytes of the function in the DLL file to point to the hook function or by replacing the address of the original function which lies on the import address table of the executable with the address of our hook function. We will discuss each method in detail and list their advantages and disadvantages in the next section.

3.3.1 The mechanism of Interception

The mechanism of interception is based on intercepting events directed at some application before they reach that application. This kind of interception is performed by functions called *filter functions*. Every filter function has the ability to modify, discard, or forward events to the intended application without changes.

The process of calling the filter functions is based on attaching these functions to a hook (i.e. setting a hook function). The hook can have more than one filter function in which it keeps a list of these functions to form a stack. That is, the most recent function is at the top and the least recent function is at the bottom. When the event occurs, the first hook function in the list in the filter functions (filter function chains) is called.

There are two main categories of hooks:

1. Intercepting Hook, and
2. Code Modifying Hook

In the next section, we will describe how they work and present the advantages and disadvantages of each method.

3.3.2 Intercepting Hook

Intercepting hook consists of two techniques to intercept API function calls. These techniques are a system-wide hook and proxy DLL hook.

The fist technique to intercept API function calls uses a system-wide hook. The system-wide hook [97][62] is based on using the three functions *SetWindowsHookEx()*, *UnhookWindowsHookEx()* and *CallNextHook()*. Windows systems, especially the Graphical User Interface (the GUI part), are based on message handling mechanisms. The events are sent to the program as messages to be processed as shown in Figure 3.2. SetWindowsHookEx() is used to set a hook inside a hook chain. One of its parameters is a pointer to the address of the filter function. These functions are called callback functions and capture messages sent to the program before it reaches the processing part and perform certain actions over the intercepted messages. UnhookWindowsHookEx() is used to remove a filter function from the chain. CallNextHookEx() is used to pass information to the next hook procedure in the chain. SetWindowsHookEx() is a common technique to intercept window messages

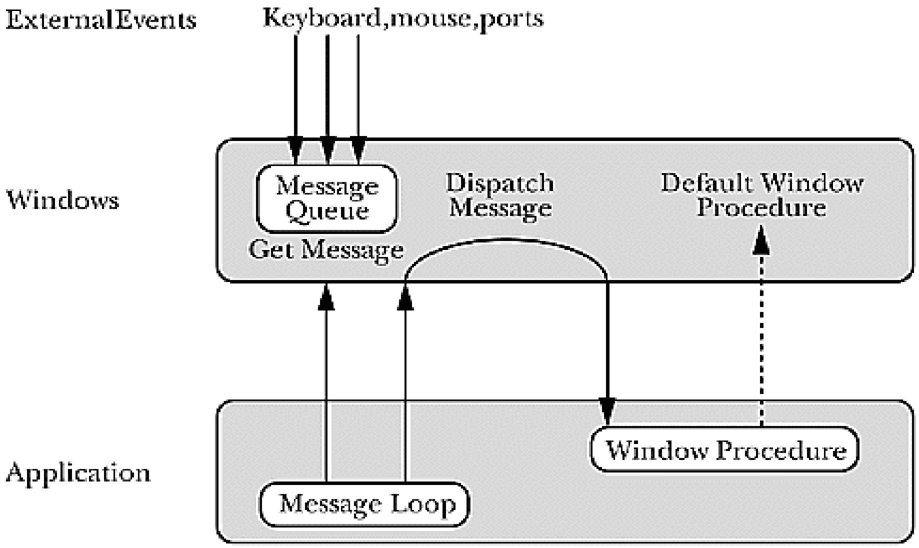

Figure 3.2: Message Handling in Windows Environment.

but it cannot intercept other function calls which do not use message handling mechanism.

A system-wide hook consists of fifteen types of windows hook (WH_*), named after their tasks (See Appendix C.1). To use a system-wide hook, these functions must reside inside a *DLL* (Dynamic Link Library). The reason for having a filter function inside a DLL is to allow the filter function to be in any application space once the hook message filter function is set. This happens because the Windows operating systems maps the DLL automatically into all the application's address space. Thus, our filter function is called for every process.

In our work, one task is to hook some functions such as *GetKeyState()*, *GetKeyboardState()*, and *GetAsyncKeyState()* used by bots which implement the keylogging activities. For example, Spybot uses GetKeyState() and GetAsyncKeyState() to record the user keystrokes. However, we found that using this kind of hook which uses SetWindowsHookEx() does not suit our requirement since the Spybot program is not a GUI program and does not have a message handling mechanism. Therefore, we implement another type of hook to handle this problem.

The second technique to intercept API function calls uses proxy Dynamic Link Library (DLL) hook [41][62][88]. This is done by replacing the original DLL with a user made proxy DLL which contains the same functions as the original file. Then, the user-made proxy DLL is copied into the target process's directory. When the program calls a function residing in the DLL, the user-made proxy DLL will be loaded and the user-defined functions will be called instead of the original functions. For example, if we want to monitor network activities, we can replace the Winsock Library (wsock32.dll) with our own fake library which contains same functions as in wsock32.dll.

Even though this method is easy to implement, it has many disadvantages. First, the program must use the proxy DLL in order to perform the intercepting process. In the previous method, the search for the DLL file is performed through the system directory, then the current directory and lastly in the PATH environment. In current method, the search for DLL file is performed in different manner. The program starts searching for DLL file in the same directory path as the program invoked, the current directory, the system directory, and lastly in the PATH environment. The second disadvantage is that the user needs to write stubs for all the exported function from the original DLL file.

3.3.3 Code Modifying Hook

Code modifying hook consists of two techniques to intercept API function calls. These techniques are hooking by entry point rewriting and import address table patching.

Hooking by entry point allows the user to change the code of the API function instead of monitoring the system messages. By doing so, the user is able to hide the process, get the system privileges and so on. In this type of hook, we rewrite the first five bytes of the function in the DLL with an unconditional jump instruction (JMP = 0xE9 in hexadecimal) to our own function instead of rewriting its IAT address on a specific file [92][88][34]. Using this method can result in the loss of the original function when we replace the first five bytes. To solve this problem, we need to save the five bytes in the trampoline function. The trampoline function contains the five bytes we replaced, plus a JMP instruction to the address of the original function, plus five.

The disadvantage of this technique is that the target function should be at least five bytes long. This is due to the fact that the jump (JMP) instruction is five bytes long. If the target function is less than five bytes long, the JMP instruction will over write some code and the target program will not be executed properly. An example of this method is implemented by the Microsoft's research development - Detours [75]. In Detours, function calls are dynamically intercepted by re-writing function using the (JMP) instruction in order to redirect to other locations.

The second technique of code modifying hook uses Import Address Table (IAT) patching. Import Address Table (IAT) patching is the process of intercepting the function calls by replacing the address of the original function with the address of the hook function [62][88][34]. To use this type of hook, the user should have good programming skills and basic understanding of Portable Executable (PE) file format. In order to understand how the hook is implemented, we will discuss the basic PE file structure and how it is created. Figure 3.3 shows the basic structure of PE file [121][122].

Every portable executable file contains different sections as shown in Figure 3.3. To implement an intercepting tool, we need to concentrate on the *.idata* section where a special table called the Import Address Table (IAT) is located inside this section. It contains a description of imported functions and their addresses from different DLL files.

Once the program is executed, the windows loader allocates memory in virtual address space and maps the executable to the allocated memory. The loader then walks through the IAT and loads every DLL file in a way similar to loading the executable. The loader then walks through the IAT of every loaded module and patches the correct addresses to every function for each module. For example, if a program needs to call a function *send()* which resides on wsock32.dll, the loader will load the DLL file to the memory space so that the IAT of that program will contain an entry to another table which lists all the functions imported from wsock32.dll including the *send()* function.

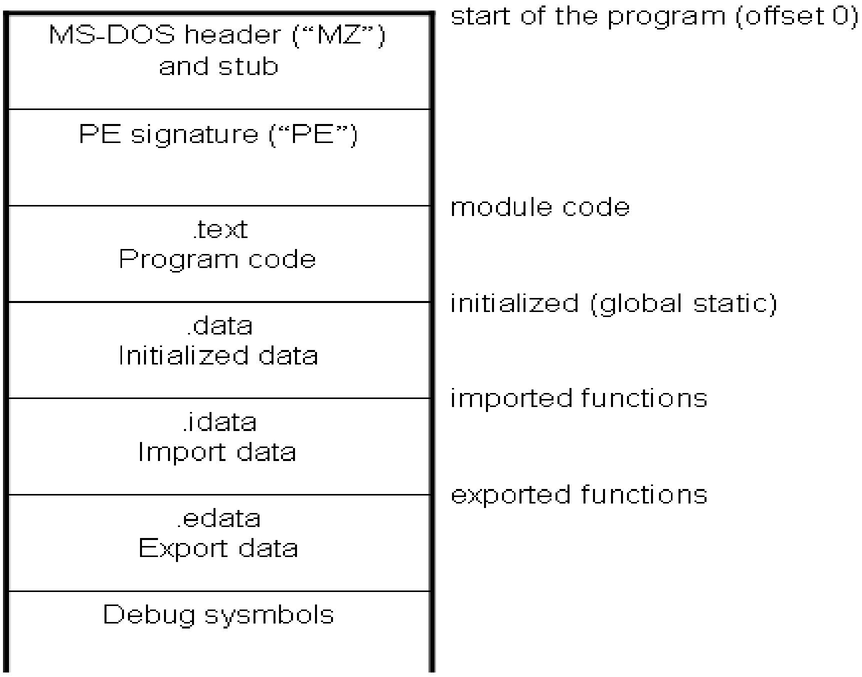

FIGURE 3.3: Portable Executable File - PE.

The interception of any function is done by overwriting this IAT. In order to hook any function, for example *send()*, we first need to store the address of the original function inside the IAT of the executable and then replace the address of the original function with the address of our hook function. This will cause all calls made to that function to be routed to our hook function. The procedure for hooking our own process is described in Appendix C.2.

3.3.4 External Process Hooking

Once the DLL is loaded into the target process, it modifies the address of the target function (e.g. recv()) in the target process so that it jumps to the replacement function in our DLL file (myrecv()). The Steps for hooking external process and DLL injection is described in Appendix C.3.

After implementing the external process hooking, we were able to hook the selected functions which are used by the bot to perform the malicious activities. For example, Spybot uses GetAsyncKeyState() and GetKeyState() function calls. Our program, APITrace, monitors these functions in order to see if the Spybot program executes these functions or not. We found that the APITrace was able to detect the use of these functions by the Spybot.

3.4 Framework for Botnet/Bots Detection

To detect a botnet or an individual bot, we designed a framework which is mainly based on monitoring API function calls executed by a process to determine if that process is anomalous or not, as shown in Figure 3.4. The framework consists of two modules. The first module is responsible for detecting the botnet while the second module is responsible for detecting an individual bot on the system. From Figure 3.4, three main API function calls are monitored, K for keylogging function calls, C for communication function calls and F for file access and registry function calls. The botnet module uses the APITrace tool to generate log files from different systems. Each file is monitored by another program which produces the change of log file data. The data is processed and analysed to detect similar changes from different log files which may indicate botnet activity.

The bot detecting module also uses the APITrace tool to monitor the same API function calls. To begin with, we use a simple correlation algorithm (i.e the Spearman's Rank Correlation - SRC) to detect an individual bot on a host by correlating different activities generated by a process. A more intelligent way (i.e. the Dendritic Cell Algorithm - DCA) of correlating different activities is used later to detect the existence of an individual bot on a system. Next, we will describe the framework in more details.

3.5 Botnet Detection through Logs Correlation

3.5.1 Introduction

We use APITrace tool in multiple systems in order to intercept the traffic between the bots and their master and save the intercepted traffic in log file on each host and then apply the log correlation technique in order to detect similar activities from multiple

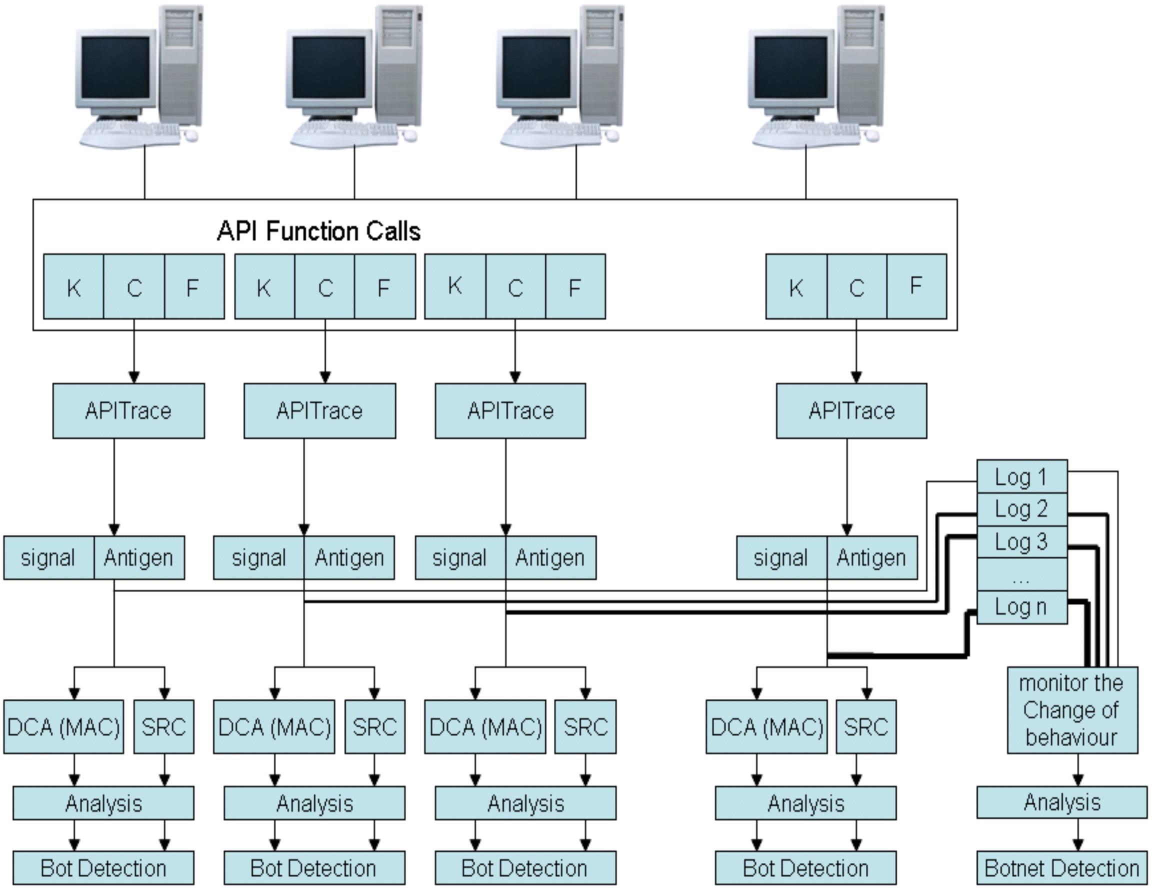

FIGURE 3.4: Framework for detecting botnet/bot.

hosts. The aim of using log correlation is to develop a host-based algorithm that is capable of detecting similar activities of similar type of IRC bots by monitoring the change of behaviours in log file sizes across several hosts and find the correlation between these changes. The detection technique is based on monitoring the change of behaviours from one state to another state in each host and observes the common actions or responses generated by bots in all hosts. This is due to the fact that bots are responding to the commands simultaneously which produce the same rate of change in each log file. The advantage of applying this approach is that this detection technique does not require searching for specific patterns when analysing network traffic. Therefore, the amount of processing time required to detect botnets is reduced. In addition, this technique does not monitor standard ports and can deal with the encrypted traffic, because this approach monitors the change of behaviour in the system not the content of each packet.

3.5.2 Input Stage

Our main goal is to detect botnets by monitoring selected API function calls from different hosts, store the collected data of log file sizes from different hosts and find the correlations between these log files. To achieve this goal, we use the APITrace tool to intercept API socket function calls executed by processes which use the communication sockets to send traffic over the network. Any other processes which do not use the communication sockets are not intercepted. The intercepted data are stored into a log file where a change in the log file is recorded by another program. This procedure is implemented in every host. For example, if a process in host (A) executes one of these function calls send(), sendto(), recv(), recvfrom(), socket() or connect(), the executed function call is stored with its arguments into a log file. In the mean time, another program is used to record the change of the log file size from the current state to the previous state every second.

3.5.3 Analysis Stage

After that, we pass the recorded data from different hosts as an input, to our correlation program. The correlation algorithm examines the generated values from each host in a sequential manner. Once we have enough correlated data which exceeds the threshold, the algorithm produces the output results.

3.5.4 Output Stage

The correlation program examines the changes between the recorded data and generates an alarm if correlated data is noticed. Correlated data represents similar activities in our network which indicates suspicious activities. Three scenarios are presented while monitoring the change of file sizes as shown:

- No file sizes changed within a specified time period which indicates that no activities happening during this time.

- All file sizes changed at a similar rate within a specified time period which indicates botnet activities responding to commands generated by the attacker.
- Different file sizes generated at different rates of change within a specified time period which may indicate normal activities.

3.5.5 Assumptions

We assume that the bots are already installed on the victim hosts, through an accidental 'trojan horse' style infection mechanism. In this case, we are not preventing the initial bot infection but limiting its activities whilst on a host machine. This means that we use an extrusion detection system instead of using intrusion detection system to detect existing bots. For our experiments, we have three different users emulated by one user on three different hosts. These hosts exist within a small virtual IRC network on a virtual machine work station (VMware). Increasing the number of hosts will result in network delays as we already examined this situation by having five virtual hosts in one physical host. In addition, because one user emulates the three uses in real time, switching between these machines could result in slow users' responses. As a result, three machines were selected to represent the best case scenario where we have fast response of users and avoid the network delays (point 54). The overview architecture of a host-based botnet detection is shown in Figure 3.5.

The three users represent the normal behaviour scenario in which they connect to IRC server and join a specific channel for chat conversation. In each host, the APITrace is running to monitor the behaviour of the three users by intercepting and logging executed function calls. For the attack scenario, we assume that the three hosts are already infected by bots through opening email attachments that contain the malicious file or by visiting a malicious website. These bots are running on victim hosts, connect to IRC server and join a predefined channel waiting for the botmaster commands. Thus, we are not trying to prevent these bots from infecting the machines but rather use an extrusion detection system, which monitors outbound network traffic to detect malicious behavioural activities.

We also assume that the files generated by APITrace are protected through en-

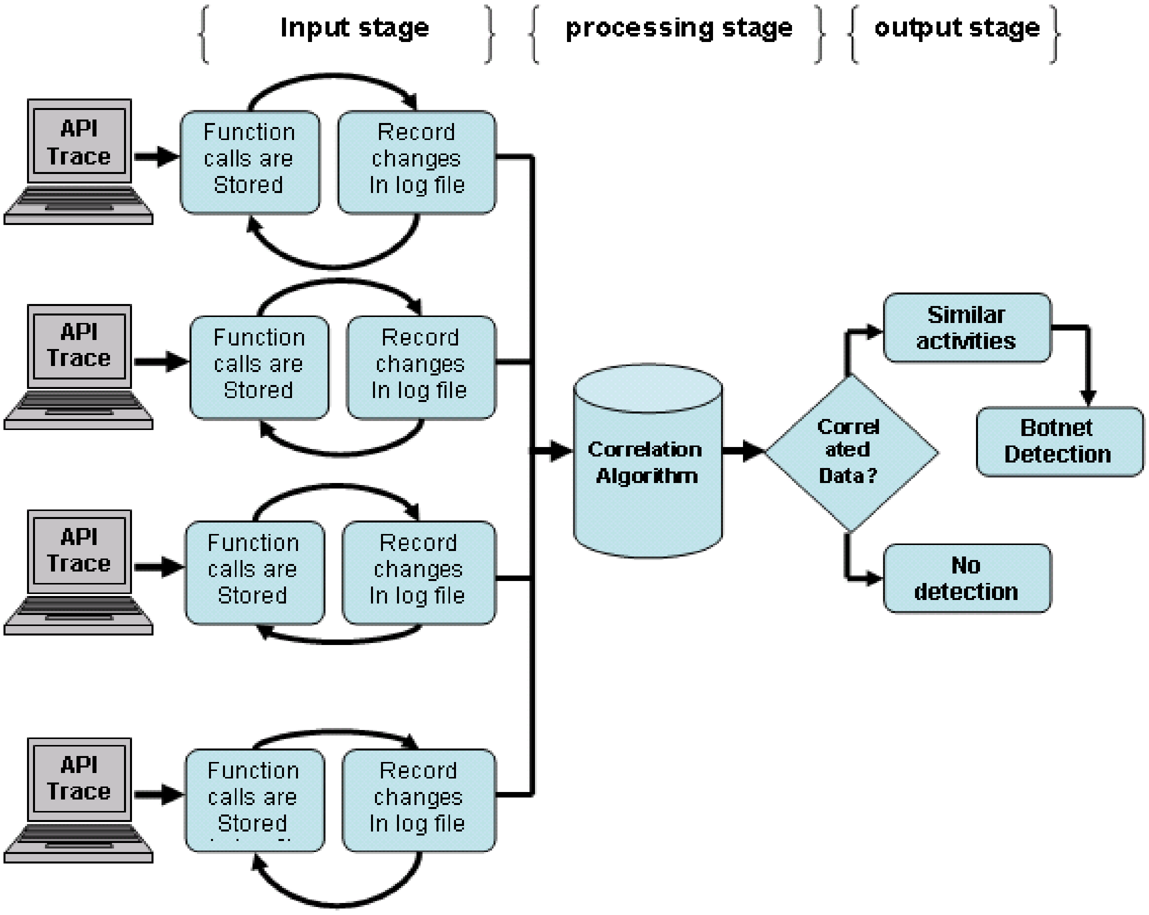
Input stage
processing stage
output stage
API Trace
Function calls are Stored
Record changes In log file
API Trace
Function calls are Stored
Record changes In log file
API Trace
Function calls are Stored
Record changes In log file
API Trace
Function calls are Stored
Record changes In log file
Correlation Algorithm
Correl ated Data?
Similar activities
Botnet Detection
No detection

Figure 3.5: Botnet Model for detecting bots.

cryption techniques or hidden from the attacker. We also assume that the attacker cannot modify or delete these files remotely as they require administrator privileges.

3.5.6 Limitation

Although this method is very useful when the communication between the botmaster and the bots is encrypted and does not require traffic analysis looking for common bots signatures but there are many issues when applying this technique. The first issue is that the botmaster can allow his bots to respond in different time periods. As a result, this technique can be defeated because it depends on the correlation of events within specified time-window. Another issue is that this method is applied where no user activities are performed as at the current stage, we do not take into account the amount of changes when monitoring the file sizes. Therefore, if a user is performing other activities such as downloading files or sending files, this will have a negative impact on the correlation algorithm. The third issue is that the number of hosts needs to be monitored when using this technique. Monitoring small number of hosts can generate false alarms if the infected host is not part of the monitored network. Another issue is that the location of the monitored hosts can play a vital role in this kind of correlation. These hosts should be targeted by the attackers to be part of the monitored hosts pool. Yet another issue is how to protect the log files from being attacked or manipulated by the attacker? One way is to encrypt the data of these files using encryption algorithms. This technique can be used to detect multiple hosts infected by same type of bots but it will fail to detect similar activities if these hosts are infected with different types of bots. In that case, a different approach is needed because our log correlation algorithm can only be applied to detect same types of bots from multiple sources.

In order to solve some of these problems, a more intelligent correlation algorithm which can correlate similar events within different time-window is needed. This correlation algorithm will be addressed in this research.

3.6 Bot Detection through Activities Correlation

Many existing techniques examine one type of malicious activities to detect the presence of the bot [26][153][44]. For example, analysing network traffic looking for suspicious patterns, monitoring port number, CPU usage or memory usage may indicate abnormal behaviour but it will generate large number of false alarms. To reduce the false alarms and improve the detection accuracy, we have decided to combine evidence of malicious behaviours such as keylogging activities which are used by many bots, sending large amount of traffic, generating large number of network errors and accessing files or registries. We correlate this evidence to increase the detection accuracy.

We represent these activities as an execution of function calls according to the following manner:

- **Communications Activities (CommFunc)**: The communication functions such as *socket*, *send*, *recv*, *sendto*, *recvfrom*, and *IcmpSendEcho*, represent an interaction between the malicious process (in our case the bot) with its master. The bot needs to receive commands from its master, execute these commands and then take an action. The bot can either send information to its master or attack other hosts or networks [112]. We can also monitor the reaction of the processes when they receive data from the network or when they send data over the network using the communication functions. It is known that the bots react faster than humans when they receive information from the network (e.g. chat conversation).

- **File/Registry Access Activities (FileAccess)**: The file access functions such as *CreateFile*, *OpenFile*, *ReadFile*, and *WriteFile*, represent the ability of bots to store some information on the host or access some sensitive information on the host by changing or deleting files and the registry [108].

- **Keylogging Avtivities (KeyboardState)**: The keyboard status functions *GetKeyboardState*, *GetAsyncKeyState*, *GetKeyNameText*, and *keybd_event*, represents the ability of the bot to perform key strokes logging to steal user information or sensitive information such as passwords, credit cards or bank account details [110].

Our main task is to develop a host-based detection algorithm that is able to correlate different activities within a specified time period to detect the malicious activities of abnormal behaviour of any process running on the host, mainly IRC bots. These activities, described above, are represented by function calls executed by the processes and intercepted by the APITrace. Instead of monitoring all function calls, our focus is to examine the behaviour of some selected function calls which we believe that they are used by most of the existing bots. The selection of these activities is based on the preliminary observation of bots behaviours. The detection algorithm can tolerate and detect different types of bots not just one type of bots. Note that this algorithm only detects malicious activities or processes on the system and does not consider any response or reaction against these processes.

Different correlation algorithms have been used to achieve this task. We start with a simple correlation technique, named the Spearman's rank correlation (SRC), to examine the behaviour of different activities of the processes, mainly IRC bots. This technique is used to measure the strength of the relationship between two activities. Although it is a simple method of correlation, it is widely used as a correlation technique.

Then, we apply a more intelligent correlation algorithm, termed the dendritic cell algorithm (DCA), to detect the IRC bots. We compare the results generated by the correlation techniques and we evaluate the strengths and the weaknesses of each technique. We modify the old anomaly value termed mature context antigen value (MCAV), used in the intelligent algorithm , and present our new anomaly value termed MCAV antigen coefficient (MAC) to increase the detection sensitivity and reduce the false positive alarms generated by the MCAV value.

In addition, we use this intelligent technique to detect Peer-to-Peer bots in a similar manner to IRC bots detection. New activities are monitored to achieve this task in addition to the previous activities used to detect port scanning. This is because that previous monitored activities are used as an inverse of each other. Therefore, we examine the effect of using different activities, which are not related to each other, on the performance of the detection algorithm. Finally, we evaluate our findings with existing techniques to measure how well this algorithm performs.

All experiments that we conducted for both algorithms, the SRC and the DCA, are performed in a small virtual IRC network on a VMWare workstation. The VMWare workstation runs under a Windows XP P4 SP2 with 2.4 GHz processor. The virtual IRC network consists of two machines, one machine runs Windows XP Pro SP2 and it is used as an IRC server. The other machine runs Windows XP Pro SP2 as an infected machine with the bot. Two machines are sufficient to perform these experiments as one host is required to be infected (i.e. the monitored machine) and the other to be an IRC server to issue commands to the bot in question.

3.7 IRC Bot Detection using Spearman's Rank Correlation

Nowadays, most bots are implemented with keylogging features. Keylogging is a means of intercepting and subversively monitoring users' activities such as typing keystrokes and mouse clicking. The intercepted keystrokes are either saved to a log file or sent directly to the botmaster. The log file can be sent to the attacker through *email*, *ftp* or accessed remotely by the attacker. Other new features added to the keylogging bot are the ability to capture screen shots and mouse logging [76].

Keylogging represents a serious threat to the privacy and security of our systems. This is because the keylogger program can collect the user's personal information, passwords, credit cards details or other sensitive information. Unlike other attacks performed by the bot, keylogging is difficult to detect as it runs in hidden mode. Many Anti-Virus packages cannot detect a stealthy keylogger running on the system. The user has no way to determine if his machine is running a keylogger, therefore, he could easily become a victim of identity theft.

To the best of our knowledge, no attempt has been made within bot detection research to detect a single bot by monitoring the Application Programming Interface (*API*) function calls except for Stinson [151]. In this section, we present an algorithm to detect a single bot in the system by monitoring and correlating different bot behaviours. We use APITrace tool to intercept specified API function calls executed by the bot which are used to perform multiple activities, mainly the keylogging activity, within specified time period. Invoking these functions within specified time window might represent a security risk to computer systems. For example, calling *GetKeyboardState* or *GetAsyncKeyState* by a program and writing data to a file using the *WriteFile* function call usually indicates a keylogging activity. In addition, the bot is designed to send the intercepted keystrokes to the attacker, therefore, we may notice a large volume of outgoing traffic during this period. Correlation of the frequency of function calls generated by the bot during a specified time-window could indicate abnormal activity in our system. We focus on three types of bot behaviour: keylogging activity, file access and outgoing traffic. By tracking and correlating these abnormal behaviours, the process of bot detection will be enhanced.

We start with a Spearman's rank correlation (SRC) algorithm to detect bots by monitoring the selected API function calls which represent activities of a bot running on the system. We perform the detection by correlating behaviours of processes based on the frequency of function calls executed by malicious processes within a specified period. SRC is a statistical measure of correlation which uses threshold function to describe the relationship between two variables. SRC is a non parametric test and does not depend on the assumptions about the frequency distributions of any variable. SRC is used to test the null hypothesis in which it assumes that there is no association between the two variables.

In comparison to other correlation techniques such as Pearson correlation, SRC performs the correlation on the ranks of the data rather than the actual data which reduces the distortion generated by Pearson correlation. In addition, Pearson correlation is used to measure the strength of linear relationship for parametric data. In contrast, SRC is used to measure the strength of relationship for non-parametric data. In comparison to Kendall's Tau, both algorithms are considered to be equivalent with regard to the underlying assumptions in terms of ranking the data. Both algorithms use the same amount of information in the data but with different interpretation. For example, Kendall's Tau represents a probability that the observed data is in the same order or not in the same order. One disadvantage of SRC is that the interpretation is less straightforward for the coefficient when it refers to the percent of variance of the two ranks. In addition, Kendall's Tau statistical distribution approaches normal distribution faster than SRC for small number of data [18].

3.7.1 Inputs

Inputs to the SRC algorithm represent the intercepted API function calls by APITrace tool. These API function calls are generated when a bot takes an action while it is running on the system. Because we focus on a keylogging activity as a main indicator to the existence of a bot, this activity is correlated with other bot activities such as sending information and file access or registry access. For the keylogging activity, we monitor the frequency of selected API function calls used to intercept the

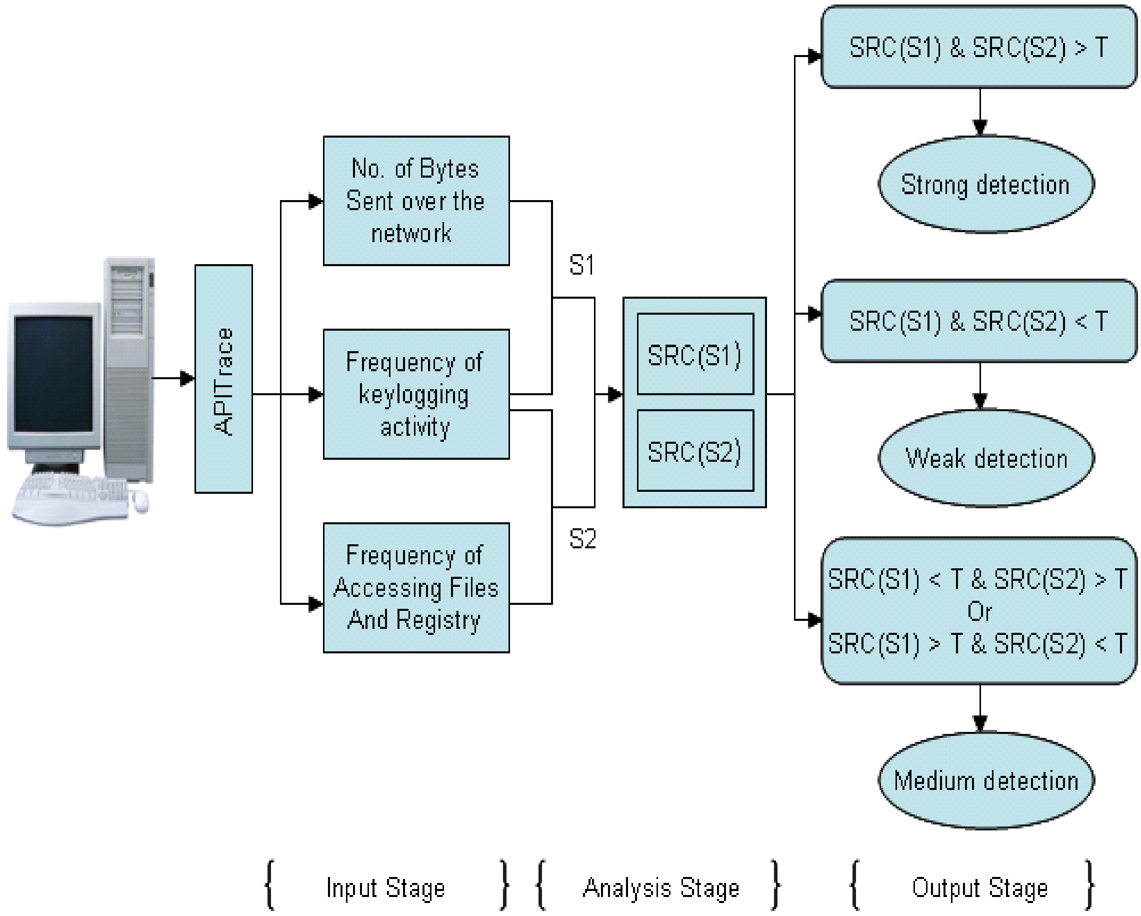
APITrace
No. of Bytes Sent over the network
Frequency of keylogging activity
Frequency of Accessing Files And Registry
S1
S2
SRC(S1)
SRC(S2)
SRC(S1) & SRC(S2) > T
Strong detection
SRC(S1) & SRC(S2) < T
Weak detection
SRC(S1) < T & SRC(S2) > T
Or
SRC(S1) > T & SRC(S2) < T
Medium detection
Input Stage
Analysis Stage
Output Stage

FIGURE 3.6: SRC Model for detecting a singel bot.

user's keystrokes such as GetAsyncKeyState and GetKeyboardState within a specified period, in our case every 10 seconds. Within the same period, we monitor (1) the frequency of API function calls used to access files such as WriteFile and ReadFile and (2) number of bytes sent on the network. We combine the keylogging activity with the file access activity to generate the first set which will be passed to the SRC algorithm for testing the two activities. We also combine the keylogging activity with the number of bytes sent over the network to generate the second set which also will be passed to SRC algorithm as you can see from Figure 3.6.

We examine two cases for each set, with zeros and without zeros. In the first case (i.e. with zeros case), the bot did not generate any activity and thus no API function call is executed. As a result, a zero value is added during that period. Having large number of zeros in our data increases the correlation between the two activities and leads to false alarms. In order to solve this problem, whenever we have a zero period for both activities, we remove that period for our data. Finally, the two sets with two cases for each set are passed to SRC algorithm to generate the correlation value.

3.7.2 Analysis

The SRC algorithm is used to find the strength of the relationship between the two activities. This is performed by generating the SRC coefficient value. The closer this value to +1 or -1, the stronger the correlation between the two activities is. The SRC algorithm first examines the existence of keylogging activity in the system. If this activity is present, the SRC then measures the strength of the relationship between this activity and other activities such as file access activity and the number of bytes sent over network activity

3.7.3 Outputs

Based on the obtained results, the algorithm will classify the situation to strong detection, medium detection, weak detection or no detection.

3.7.4 Assumptions

Different assumptions have been made when using SRC algorithm. One assumption is that the bot is already installed and runs on a victim's machine. Thus, we are not trying to prevent our system from being infected but rather detect malicious activity on our system. Another assumption is that the bot is programmed to perform keylogging activities and in order to do that, the botmaster uses specified API function calls to intercept the user's keystrokes such as GetAsyncKeyState or GetKeyboardState. Executing these functions by a process may represent abnormal keylogging activity in our system. However, we consider that calling these functions generates only a 'weak' alert because some legitimate programs may use the same API calls. Therefore, the need of collating different activities can enhance our detection algorithm to form a 'strong' alert. We also assume that our monitoring program is protected through proper encryption technique.

3.7.5 Limitations

The scope of this detection technique is to develop an algorithm that is able to correlate different malicious activities within specified time period to detect a bot running on the system by applying SRC. The SRC algorithm performs a simple correlation technique to measure the relationship between different activities of a process. This SRC algorithm can detect different types of bots and is not restricted to one specific type of bots. Although this algorithm is effective in detecting malicious behaviours, it can generate false alarms when the bot is idle, thus no activity is noticed. In addition, if the botmaster programs his bot to perform these activities within different time periods or allows his bots to work in a stealthy manner, this method can be defeated. In order to solve these problems, an intelligent algorithm is needed to correlate different activities using dynamic time-window. This algorithm will be discussed in the next section.

3.8 IRC Bot Detection using the Dendritic Cells Algorithm - DCA

Artificial Immune Systems (AIS) are algorithms inspired by the behaviour of the human immune system. The biological immune system tries to protect the body against attacks from invading pathogen, viruses and bacterias. The first generation of AIS, which have been applied in the area of computer security, did not generate a high detection level of malicious activities [148]. In addition, it lacks scalability and produces large number of false alarms. In order to overcome the problems generated by the first generation of AIS algorithms, a second generation AIS algorithm is presented.

A recent addition to the AIS family is the Dedritic Cell Algorithm (DCA) implemented by Greensmith [51]. DCA is inspired by the function of the Dendritic Cells (DCs) of the innate immune system and uses principles of a key novel theory in immunology termed the danger theory described by MAtzinger [102]. The danger theory suggests that the DCs are the first line of defense against invaders and the response is generated by the immune system upon the receipt of molecular information which indicates the presence of stress or damage in the body.

The DCA performs multi-sensor data fusion on a set of input 'signals' which reflect some activities in our host, and this information is correlated with potentially anomalous 'suspect entities', termed antigen. These antigen are classified into normal or anomalous. The collected data forms an input data to the DCA.

The aim of using the DCA algorithm is to detect a bot running on the system by correlating different activities (represented by signals) and trace the suspect causing these activities (represented by antigen) from multiple sources and produce the state of the entity, either normal or anomalous. The DCA will not only state if anomaly is detected but it will identify the culprit responsible for the anomaly. The DCA is applied to various types of bots and the results show that it is not restricted to one type of bots.

3.8.1 Abstract model of DC biology

In order to understand the DCA algorithm, an abstract model of the DC biology is required. DCs perform different functions such as data sampling and analysis and signals and antigen processing. The processing of input signals and antigen occurs in the tissue; a monitored environment by DCs. Once a DC has collected enough information, it migrates to a processing centre called a lymph node for presenting antigen with context signals. The results allow the human immune system to take an action if it feels that the human immune system is in threat.

DCs exist in one of the three states: immature, semi-mature and mature cell. While in immature state, DCs collect multiple antigen using a sampling mechanism. These antigen are the data to be classified either as normal or anomalous data. The process of collecting antigen does not activate the immune system. DCs also receive different input signals from different resources. If the DC receives enough input signals, it will transform to another state. The terminal state is determined by the kind of the input signals received. If the DC received more 'safe' signals, it will transform to a 'semi-mature' DC. In contrast, if it received more 'dangerous' signals, it will transform to a 'mature' DC. Based on the final state of the DCs, the immune system will respond accordingly.

DCs are sensitive to the type of signals they collect. Four types of signals can affect the transformation process. These signals are safe signals, danger signals, Pathogen Associated Molecular Patterns (PAMP) signal and Inflammation signal. These signals will be discussed in more details in Chapter 6. A high concentration of safe signals leads to a semi-mature cell while a high concentration of danger and PAMP signals leads to a mature cell. The relationship between the signals and the mechanism of signals processing is abstracted from [174].

One feature of DC is that it can sample antigen and signal simultaneously. Multiple DCs produce multiple copies of the same antigen type. This procedure allows error tolerant component as a misclassification by one DC is not enough to generate a false error from the immune system. In addition, each DC is randomly assigned a migration threshold value which adds robustness to the system.

3.8.2 Inputs

Signals and antigen are passed as an input to the DCA. To collect signals and antigen, an APITrace is used. This monitoring program generates two log files, one for signals and the second for antigen. Various signals are used which include PAMP, danger signal, safe signal and Inflammatory signal. These signals with their functionality are described below both in biological and abstract terms.

In a biological term, PAMP are molecules produced by microorganisms which represent an indication of foreign entity in the body treated by the immune system as a biological signature of abnormality. In the abstract model, PAMP is also treated as a signal for abnormality. A high concentration of PAMP input signals increases the costimulation output signal (CSM) and the mature signal (mat) produced by the artificial DCs in the abstract model. In the algorithm, the CSM is used to determine the lifespan of the DC signal and antigen sampling when assessed with DC migration threshold.

Danger signals represent an indication of damage to the tissue. When a DC receive danger signals caused by an expected death of cells (termed necrotic), it transforms from an immature state to a mature state. In the abstract model, danger signals are an indicator of abnormality but have less impact than PAMP signal on DC. Danger signals also increase the amount of CSM output signals which allow the cell to transform from immature state to mature state.

In human immune system, safe signals are released as a result of normal death of cells (termed apoptosis) which represent a healthy tissue cell function. When a DC receives safe signals, it updates the CSM output signal in a way similar to PAMP and danger signals. A high concentration of safe signals transforms the immature cell to a semi-mature cell. One of the main functions of the safe signals is that they suppress the effect of the PAMP and danger signals which provides tolerance to the system and reduces the false positives made by the immune system. In the abstract model, safe signals represent normal behaviour of the system. A high level of safe signals increases the semi-mature output signal value and reduces the cumulative value of mature output signal.

Inflammatory signal in human tissue implies that the inflammatory cytokines are present and the temperature is also increased in the affected tissue which increases the rates of reaction. In the abstract model, the inflammatory signal is used to amplify the other three signals, PAMP, danger and safe signals which lead to an increase of the output signals. This causes DCs to migrate rapidly as the magnitude of CSMs produced by the DC will occur in a short period of time. As a result, the DC lifespan in the tissue will be reduced. It is important to know that the presence of inflammatory signals alone is insufficient to initiate maturation of an immature DC.

The above signals would be sufficient to indicate if the tissue is under attack but it will not provide any information regarding who is causing this attack. As a result, antigen or the suspects are needed to link the evidence of changing behaviour of tissue with the entity which has caused this change in behaviour. Because antigen represent the data to be classified in our model, more than one type of antigen is required.

In our work to detect IRC bots using the DCA, PAMP signal is used to indicate the keylogging activity in our system. The danger signal is used as an indication of fast reaction between the bot and the botmaster. The safe signal is used to indicate how fast the bot invokes the same communication function calls. The inflammatory signal is set to a zero which indicates that it does not have any effect on the other three signals. We use selected API function calls described in section 3.6 with their process ID to represent our antigen.

The signal selection for detecting bots is based on the assumption that most of the bots exhibit similar behaviour and perform similar task. For example, bots perform keylogging activities and steal sensitive information, launch distributed denial of service attacks which leads to large amount of traffic sent over the network and access files and registry. In addition to these activities, peer-to-peer bots search for other bots over the network which may produce messages to indicate failed attempts. These common activities of the bots and the consequences of their attempts can be used as our signals.

A libtissue framework [167] is used for developing an immune inspired algorithm and to provide an environment which creates DC objects and processes the input signals as shown in Figure 3.7. Libtissue is an API library implemented in C which is based on the principles of innate immunology and used within the danger project [1] to assist the implementation and testing of immune inspired algorithms on real-time data. This framework allows implementing algorithms as a collection of cells, antigen and signals.

Libtissue is a client/server model. The communication between the client and server is performed via sockets using Stream Control Transmission Protocol (SCTP). The client is responsible for processing the input raw data and transforming it into antigen and signals as shown in Figure 3.7. The client is responsible for passing the input data (antigen and signals) to the tissue servers. Each tissue server is responsible for storing a fixed size antigen and level of signals set by the client. Cells, signals and antigen interact in the tissue server. The server is used as an algorithm to provide the DCA components and parameters.

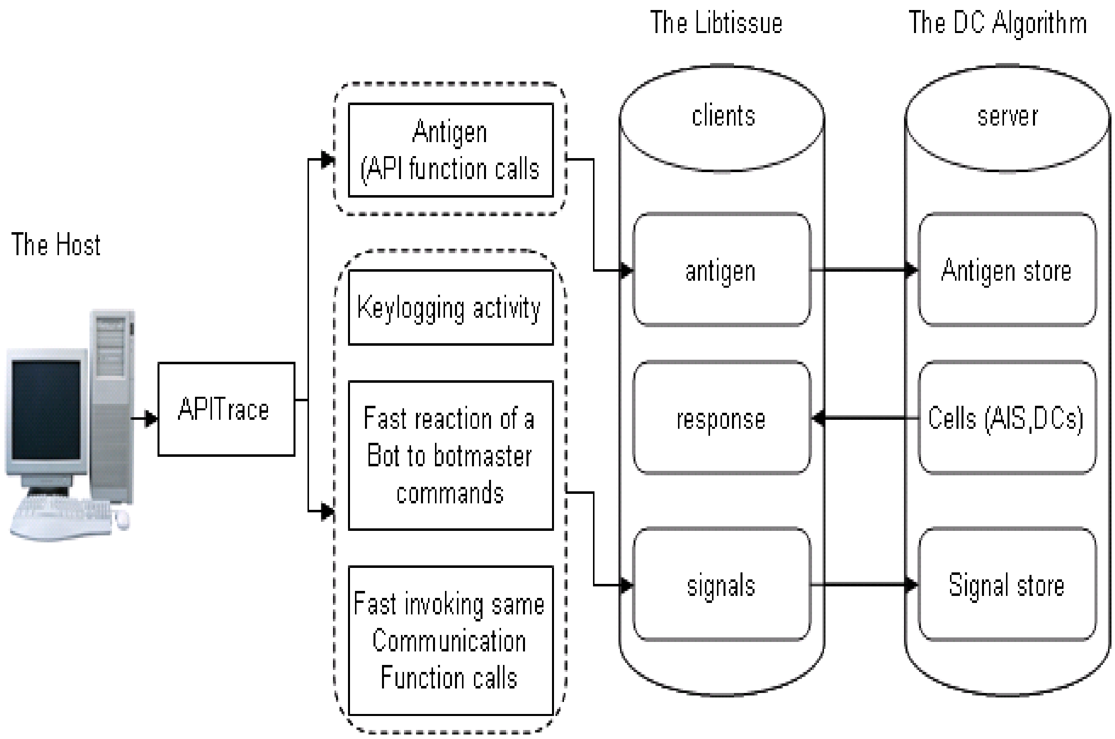

FIGURE 3.7: Server-Client model to support the DCA. The input signals and antigen are collected by the monitoring program and it is passed to the server using the client.

3.8.3 3.4.7 Analysis

The input signals (PAMP, danger, safe and Inflammatory) are pre-normalized and pre-categorized to reflect the behaviour of the system. These signals are combined and sorted with the collected antigen in a timely manner and passed to the algorithm using the libtissue.

In the algorithm, all DC based algorithms are population based. Each DC within the population performs its own signal collection and antigen sampling. In addition, each DC is assigned a random migration threshold to terminate its signals and antigen collection. This allows each DC to sample different data within different time window which adds robustness to the system. The range of the migration threshold is a user defined parameter for all DC population. A median point is used to represent the value between the lower and the maximum value of the input signals and the range of the random assignment is + or - 50% of the median value. The derivation of this range is described in details in Greensmith Thesis [46].

After receiving signals and collecting antigen, each immature DC processes the received signals to produce a set of cumulative output signals as you can see from Figure 3.8. The signal processing is performed through a simple weighted sum equation (Equation 3.1) to reduce any additional computational overheads.

$$O_j = \sum_{i=1}^{3} (W_{ij} * S_i) \qquad \forall j \tag{3.1}$$

The weight values are derived empirically from immunological data and experiments performed on natural DCs which produce good results when applying sensitivity analysis for a given application. The actual values used for the weights can be user defined except for the semi-mature weights where it must be constant (i.e. $W_{pamp} = 0$, $W_{danger} = 0$, $W_{safe} = 1$). The generated output from Equation 3.1 are $O_1 = CSM$, $O_2 = semi$ and $O_3 = mat$. These outputs are cumulatively summed over time. The CSM is assessed against the DC migration threshold to limit the duration of DC signal and antigen sampling and collection. Once the CSM cumulative value exceeds the migration threshold, the DC terminates its lifespan and it is removed from the sampling area in the lymph node. The cell then presents its antigen and output signals it has collected during its lifespan. If the cell exposed to more safe signals during its lifespan, it transforms to a semi-mature DC and presents antigen with context value of zero. In contrast, if the cell is exposed to more danger signals, it transforms to a mature DC and presents antigen with context value of one. These context values are used to form the anomaly detection. A detailed description of the algorithm will be discussed in Chapter 6.

3.8.4 Outputs

For the DCA, DCs produce three output signals as a result of exposure to the input signals. These three signals are the mature (mat) output signal, the semi-mature (semi-mat) output signal and the costimulatory molecules (CSMs). In the human immune system, the CSM is combined with another receptor to attract DC to the lymph node for antigen presentation. In order to simplify this process, a simple version

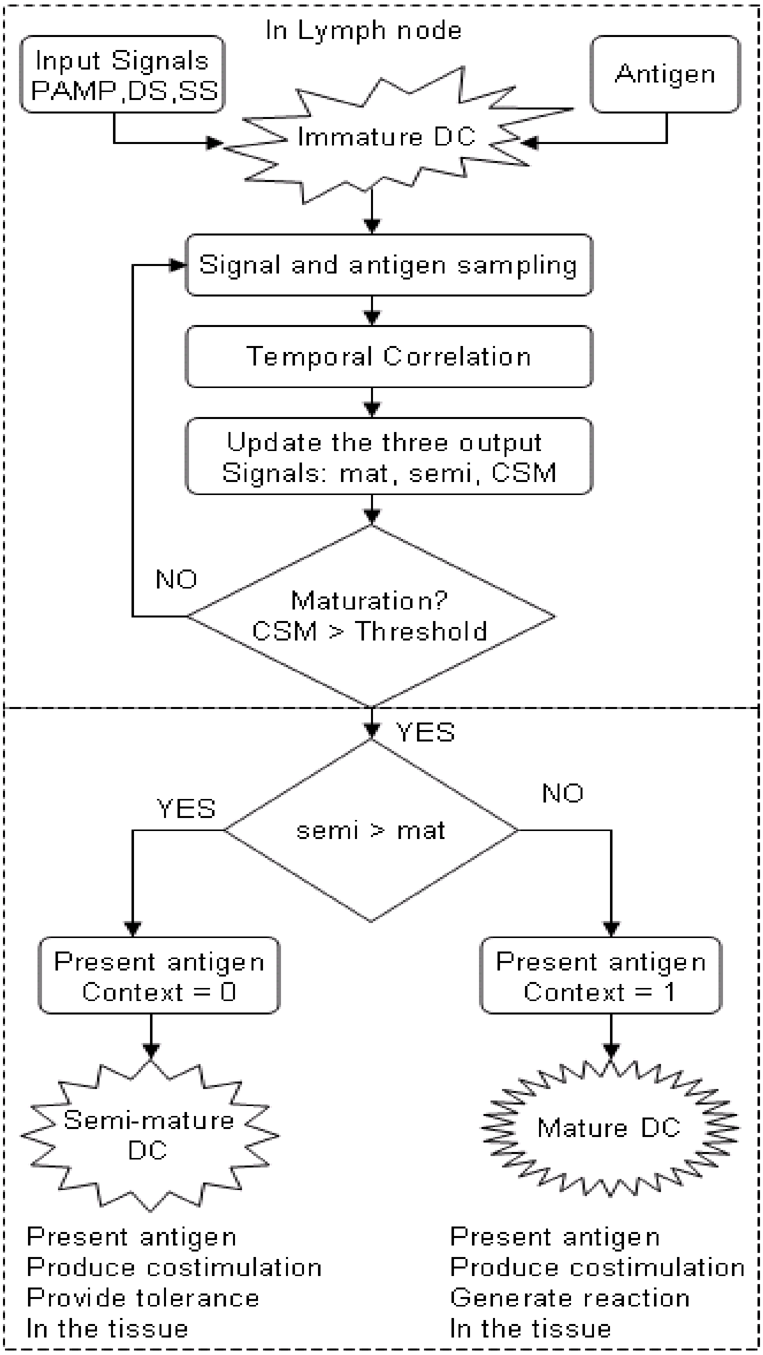

FIGURE 3.8: Abstract model of the DCA.

is used in the algorithm to achieve a reasonable result. In the abstract model, a high value of CSMs increases the probability of a DC moving from tissue to a lymph node for analysis. Because every DC is randomly assigned a migration threshold when it is created, exceeding this threshold allows the DC to change its state from immature DC to either mature or semi-mature DC. The cell then presents its antigen where its context is assessed. The context of the DC is presented by the relative proportion of semi-mature cells to mature cells. A large number of semi-mature cells indicate normal activity for the process. In contrast, a large number of mature cells indicates abnormal activity for the process. This is presented by anomaly coefficient score, termed MCAV, from zero to one. The closer the value of the anomaly score is to one the high probability that this process is anomalous. One problem of this anomaly coefficient is that it produces false positive error when the number of antigen to be classified is relatively small. In order to solve this problem, we have introduced a new anomaly coefficient value termed MAC value. The advantage of the MAC value is that it takes into account the total number of antigen being classified for all processes.

3.8.5 Assumptions

Various assumptions are made to simplify the model. It is assumed the cells do not communicate with each others. In addition, only four signals are used in this model and the DC does not take into account any other signals. For the output signals, only three signals are used to generate the output signals. Furthermore, only a single tissue is used for simplicity. Moreover, no other types of cells are used in the implemented algorithm.

3.8.6 Advantages and Limitations of DCA

In comparison to other machine learning techniques such as decision tree classifiers, neural networks, support vector machine and genetic algorithms, the DCA is unsupervised learning algorithm which does not require training phase on normal data. It classifies the data into normal or anomalous based on the categorization of signals. In addition, it is applicable to work high dimensional datasets. The DCA has many novel ideas. One of these novelties is that the algorithm has a unique method of combining multiple signals and correlating the values of these signals simultaneously with antigen to provide a tolerance to the system by using the safe signal which suppresses the effect of the PAMP and danger signals to reduce the false alarms. This process of combining signals and correlating it with the suspect antigen makes the system difficult to compare with other standard machine learning techniques mentioned previously or signature based IDS because of the types of the data being used [52][49][58]. As a result, we compare the DCA performance with other correlation algorithms such as Spearman's Rank Correlation because both algorithms perform correlation between different datasets.

Another important novelty about the algorithm is that each DC sample antigen and signals at different time windows which provide robustness to the system. Furthermore, in contrast to other AIS models, the DCA does not use any pattern matching techniques and does not require any training phase before applying to some application areas. Moreover, the DCA can use more than one signal from the same type. For example, the DCA can use $PAMP_1$, $PAMP_2$ $\ldots PAMP_n$ as different PAMP signals. These signals can also be combined to form an average value for PAMP signals. For example:

$$PAMP = \frac{\sum_{i=1}^{n} PAMP_i}{n} \tag{3.2}$$

This feature of the DCA provides user flexibility in the selection of signals. If one signal from the same type did not work for certain reason, the other signals can compensate the effect of that signal. This flexibility of having more than one signal from the same type has many advantages. For example, it can reduce the effect, when the user does not know if one PMAP constitutes strong evidence. It allows the DCA algorithm to be applied to different types of bots without changing the signals.

Although DCA has novel features, there are few points to consider when using the DCA algorithm. First, the number of antigen for each process should be enough in order for the DCA to make a proper classification of processes. Few numbers of antigen can generate poor performance of anomaly coefficient and leads to misclassification of the processed data. In order to solve this problem, we have modified the existing anomaly coefficient by taking into account the total number of antigen for all monitored processes. In addition, for highly active processes which are invoked simultaneously, the DCA will suffer from the difficulties of discrimination between the two processes and may classify these processes as anomalous processes. In case the DCA considers a benign process as a bot by giving it a higher anomaly score, this situation will generates false alarms and might be because this process is highly active for some reasons such as malfunctioning by trying to make connections to other parties, sending large amount of information or downloading large files. Moreover, If new bots appear, there is no need to redesign the DCA but instead a proper selection of signals and their weights beforehand can increase the performance of anomaly detection. These signals should be carefully chosen because the algorithm does not require any training phase. A poor selection of signals and weights can reduce the performance of DCA and may generate false alarms. In this situation, the DCA may either classify normal process as malicious process or classify malicious process as normal process. Therefore, a proper characterisation of the input signals and applying adaptive signals and variable weights can increase the performance of the DCA. Although this action can improve the detection scheme, an adversary can defeat the DCA by allowing his bots to act in a stealthy manner. The botmaster can design his bot to delay the bot's response to the commands issued by him or can transmit traffic in different time periods, thus changing the behaviour of the bot. This lead us to choose different types of input signals. The adversary can also defeat the DCA by finding out the weight ratio of the signals from research papers and try to change the weight ratio in the DCA to produce normal activity. In addition, the adversary can possibly find out which attributes are used for which signal. Once the adversary knows this, s/he can ensure that by adding enough noise/data to all of the signals, the process behaviours will be changed and appear as normal. For example, if CPU is assigned as a danger signal and the safe signal is represented by networking activity with a pre-defined weights for both signals, whenever the adversary wants to issue an attack s/he needs to make sure that he adds enough noise to each signal, to make the ratio of danger signal vs. safe signal always result in a normal signal and thus, the generated anomaly value will be always low.

For new bots which exhibit different behaviour such as a bit torrent-based bot, the selection of signals should be adaptive by having a range of signals from same category to deal with this situation. In that case, the DCA can have multiple input signals as presented in [49] for SYN scan detection or these signals from the same category can be combined to form one signal as we have done when applying DCA to detect P2P bots. If the behaviour of a new bot is not similar to behaviours previously considered, the DCA might miss classify this process and thus, generates false alarms. Attacking the monitoring program or the DCA is another issue that an adversary can think about. As a result, these programs should be placed and protected where it is difficult for an adversary to figure out and attack.

3.9 P2P Bot Detection using the Dendritic Cells Algorithm - DCA

In the previous section, we use the DCA to detect IRC bots which are widely used by attackers. This is based on bots connecting to a centralized point in order to receive commands from the botmaster. When using a centralised structure, one can prevent the bots from communicating with their masters by shutting down the central point [37]. Recently, the botmasters started using another type of command and control (C & C) structures know as Peer-to-Peer (P2P) networks in order to control their bots. In contrast to IRC bots, P2P bots contact other peer bots without referring to a centralised point. This approach provides an efficient way of controlling bots to maintain the bots functionality.

Similar to detecting IRC bots using the DCA, we will also use the DCA with pre-defined signals in order to detect the running bot. These signals represent our inputs to the DCA which are derived from monitoring the behaviour of the P2P bots. Three signals have been considered for detecting P2P bots. The first signal is the PAMP signal and represents a combination value of different subset signals which are: destination unreachable (DU), connection reset (RST) and failed connection attempt (FCA). Because of the nature of P2P structures, peers try to find other peers in order to communicate with. Some peers cannot be reached for certain reasons and this generates large number of DU, RST or FCA. For example, some peers are switched off, some peers do not exist any more and some peers are behind firewalls. Another reason could be that the time for searching for peers is too long. These kinds of error messages may indicate a malicious activity in our host. The second signal is the danger signal. Because P2P bots generate high traffic volume when searching or communicating with other peers, we have considered this activity as our danger signal. The last signal, is the safe signal. The safe signal is the same activity for detecting IRC bots which is how fast the bot executes same function calls. This activity is important when the bot is designed to participate in a denial of service attack. Other PAMP signals could be used for this case. This includes the number of connections made by bot per second because the bot will try to connect, to many peers within a short time period. Another PAMP signal which has to be considered is the CPU and memory usage. For the time being we only concentrate on the three signals that are described previously.

The analysis, the output stages and the assumptions are similar to the IRC bots experiments described in the previous chapter. Two bots (Phatbot and Peacomm) will be examined in our detection experiments as a case study.

3.10 Summary

In this chapter we discussed different interception techniques to monitor API function calls. As a result, we have developed APITrace tool to monitor specific function calls which are used in malicious programs. We also give an abstract design of our framework to detect the botnet/bot which is based on monitoring API function calls in Windows operating system using APITrace. In addition, we present in detail different methods of detecting botnet/bot by correlating different activities on the system and discuss how we collected data. A more intelligent way of correlating these activities using DCA algorithm is also presented with a modification of mature context antigen value (MCAV) to improve the detection accuracy and reduce the generation of false alarms.

Chapter 4

Host-Based Botnet Detection

4.1 Introduction

Many existing botnet detection techniques monitor the behaviour of groups of bots rather than an individual bot. These techniques use traffic analysis and pattern matching algorithms to achieve their task. Although traffic analysis and pattern matching techniques perform well for detecting known bots, they suffer from different problems such as they cannot detect new bots' patterns. In addition, they consume large amounts of time and resources while they analyse network traffic as explained previously. A new technique is suggested by Cooke et al. [26] to enhance botnet detection schemes. Instead of searching for the command and control by analysing network traffic, he suggests that the data can be correlated from different sources to locate bots and discover command and control connections, but no further details or results are presented.

To address these problems, we have started our analysis for botnet detection by monitoring the change in behaviour instead of network traffic analysis. We have implemented an algorithm that monitors the change of log file sizes across several hosts and finds the correlation between these changes. Most bots respond simultaneously to the commands issued by the botmaster. As a result, this operation generates the same rate of change in each log file. Our approach examines these changes and finds the correlation between them. Our approach does not search for botnet traffic with specific patterns. Therefore, the amount of processing time required to detect botnet will be reduced. In addition, If the packets are encrypted, our algorithm can deal with that situation.

4.2 Methodology

We use the APITrace tool to intercept API communication function calls for each process that uses communication function calls in each host. Such functions include send(), sendto(), recv(), recvfrom(), socket() or connect(). Once intercepted, each function call is stored in a log file with its arguments. Meanwhile, another program is used to monitor the change in the file size and pass the recorded data to our correlation program. The input to our correlation algorithm is the recorded data from the change in file sizes in each host which then, are passed to the correlation algorithm. The output of our correlation program is to generate an alarm if correlated data is generated. The correlated data represents similar activities in our network and may indicate suspicious activities in the form of botnet activities.

4.2.1 Data Collection

In order to collect the data, we have used the APITrace interception program to monitor the executed function calls by processes. We have used the import address table (IAT) patching as described in section 3.3.3 to perform this task. There are many reasons for choosing this method. The first reason relates to its ability to intercept both console and GUI applications which cannot be performed when using SetWindowsHookEx() method. Another reason is that we do no need to write a stub for all exported function calls to our DLL file as we have to do when using Proxy DLL methods.

4.3 Design and Implementation

Our APITrace tool intercepts the API communication functions and stores the intercepted functions with their arguments in a log file in each host. The generated log file is monitored by another program to detect the rate of change of the amount of traffic as shown from Figure 4.1. The process of our algorithm is as follows: First, we intercept API socket function calls used by communication programs such as send(), recv(), sendto(), recvfrom(), and store them with their arguments into a log file. During the first step, another program is used to record the change in the amount of traffic generated. This record is made every second for a specific time window t, in our case, 10 minutes. This time window is chosen based on preliminary experiments we have conducted in which we intercept most of the commands generated by the attacker. We assume that the log files are protected and the attacker cannot modify or delete the log files. An example of these log files is shown below:

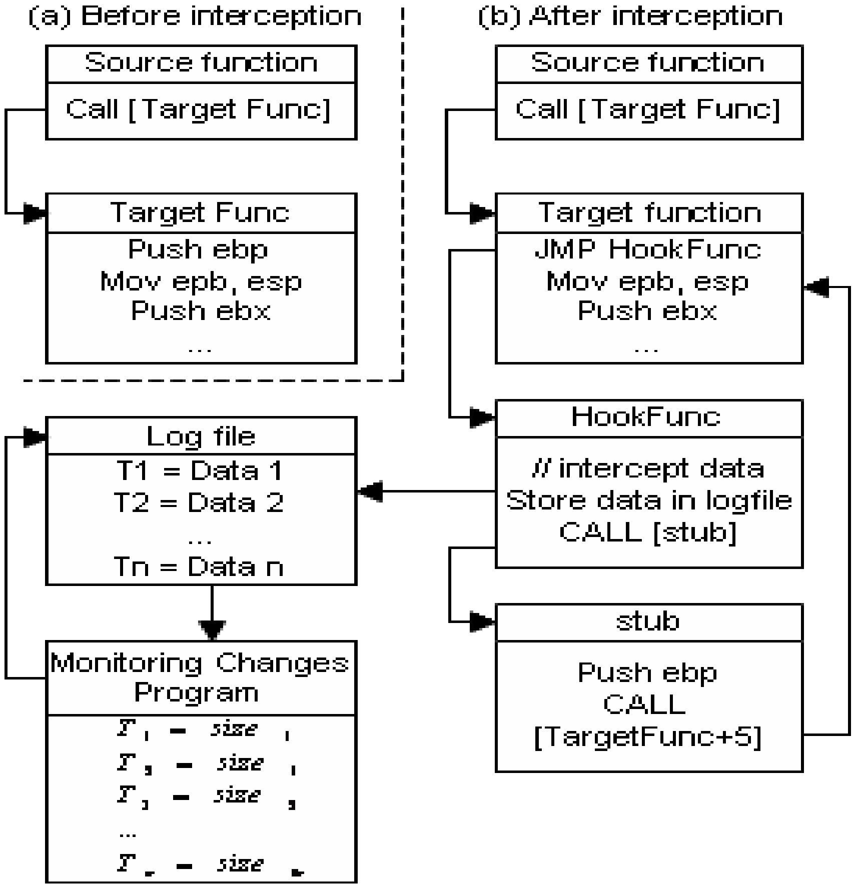

FIGURE 4.1: API function calls: (a)Before Interception vs. (b)After Interception.

```
1240331543 socket(af=AF_INET, type=SOCK_STREAM)
1240331544 connect(SOCKET=256, sockaddr=8df40c)
1240331550 send(SOCKET=256, size=42)
1240331553 recv(SOCKET=256, size=998)
1240331558 send(SOCKET=256, size=42)
1240331560 recv(SOCKET=256, size=998)
1240331566 send(SOCKET=256, size=42)
1240331578 send(SOCKET=256, size=42)
1240331579 recv(SOCKET=256, size=998)
1240331582 recv(SOCKET=256, size=998)
1240331603 closesocket(SOCKET=164)
```

After a time window t, the recorded data is passed to the analyzer which is used for correlating data. The analyzer reads the recorded data for each host and checks to see if there is a change in the current state (e.g. time *t2*) as compared to the previous state (e.g. *t1*) for all recorded data from different hosts. The output is *ones* if there are changes between all log file sizes and *zeros* if there are no changes between log file sizes. Both all zeros (i.e. no change between log files made from different hosts) and all ones (i.e. all log files from different hosts are changed) mean correlation between data is as shown in Table 4.1.

For example, if there is no change between datasets at time t_1, (logfile1=0, logfile2=0, logfile3=0, ...), then we have *zeros* change correlation. If there are changes in all datasets at time t_1, (logfile1=1,logfile2=1,logfile3=1,...), we have *ones* change correlation. Otherwise, an uncorrelated event is recorded. Our correlation algorithm is shown in Algorithm 1. The effect of changing correlation threshold will be examined with ROC curve while discussing our results. Using hooking techniques for communication applications reduces the need for monitoring the changes in the whole system. As a result, we will have only a few applications to monitor.

4.3.1 Full details of Architecture

To perform our experiments, we set up a small virtual *IRC* network on a VMWare machine. The VMWare machine runs under a Windows XP P4 SP2 with a 2.4GHz processor and 1GB RAM. The virtual *IRC* network consists of four machines. One machine runs Windows XP Pro SP2 and it is used as an *IRC* server. The remaining

Table 4.1: Change in Log File Sizes Correlation

Time	$LogFile_1$	$LogFile_2$	...	$LogFile_n$	Result
T_t	0	0	0	0	zeros correlation
T_t	1	1	1	1	ones correlation
T_t	0	0	0	1	no correlation
T_t	0	0	1	0	no correlation
T_t	0	0	1	1	no correlation
T_t	0	1	0	0	no correlation
T_t	0	1	0	1	no correlation
T_t	0	1	1	0	no correlation
T_t	0	1	1	1	no correlation
T_t	1	0	0	0	no correlation
T_t	1	0	0	1	no correlation
T_t	1	0	1	0	no correlation
T_t	1	0	1	1	no correlation
T_t	1	1	0	0	no correlation
T_t	1	1	0	1	no correlation
T_t	1	1	1	0	no correlation

machines run Windows XP Pro SP2 and have *IRC* clients. Different experiments are conducted to analyse normal behaviour and abnormal behaviour. Each experiment was running for 10 minutes in order to collect a reasonable amount of traffic.

4.3.2 Experiments

We conducted some initial experiments to determine if network statistics logs alone are sufficient to detect bots. For example, we monitor the change of behaviour of Internet Explorer *(IE)* vs. *sdbot* [132]. The results show that there is a sudden increase in log file size when the bot herder uses his bot to perform UDP, or ICMP flood attack against other systems. On the other hand, *IE*, which is used for browsing, checking emails, and other services not including downloading/uploading files, shows a smooth increase in log file size. After that, we have investigated the normal behaviour of mIRC clients vs. the *sdbot*. Monitoring changes of behaviour of normal mIRC clients and *sdbot* shows that there is a sudden change in the case of transferring large files between mIRC clients similar to a bot attack. In order to distinguish normal

```
forall the $logfiles$ do
    read file sizes of each $logfiles$;
    if all current file sizes did not change from the previous sizes then
        $outfile$ = generate $zeros$ correlation;
    else if all current file sizes changed from the previous sizes then
        $outfile$ = generate $ones$ correlation;
    else
        /* some current file sizes changed */;
        $outfile$ = generate uncorrelation;

end
while $!eof.outfile$ do
    if zeros correlation || ones correlation then
        $CV++$                    /* Correlated Value (CV) */;
    else                                       /* Uncorrelated Value (UCV) */
        $UCV++$ ;
    end
end
if $CV > Threshold$ then
    suspicious activity is detected;
end
```

Algorithm 1: Correlation Algorithm

behaviour of mIRC clients and abnormal behaviour of bots, we analysed two cases: the normal case and the attack case. In the normal case, we analysed two scenarios:

- Three users having normal conversations.
- Three users having normal conversations and sending files to each other.

In the attack case, we analysed two scenarios:

- Three bots join an *IRC* channel and remain idle for two minutes. After the idle period, the bots start to receive commands from their master. In this scenario, we exclude the flood attack commands from the botmaster as we want to examine the behaviour of the bots without generating large amount of data through flood attack commands.

- Three bots joining an *IRC* channel and remaining idle for two minutes. After the idle period, the bots start to receive commands from their herder including flood attack commands.

The generated results are passed to our correlation algorithm to distinguish normal behaviour and abnormal behaviour. Note that we have normalised the x-axis which represents the file sizes to 100 bytes in order to make the graphs more comparable. The next section explains our results in more detail.

4.3.3 Hypotheses

Different hypotheses are used for the conducted experiments. These hypotheses include:

1. Null Hypothesis One: There is a difference between normal behaviour and abnormal behaviour for mIRC client and the bot when looking at logs.
2. Null Hypothesis Two: There is a high correlation between the change in log file sizes for users having normal conversations.
3. Null Hypothesis Three: There is a high correlation between the change in log file sizes when responding to the botmaster commands.

These hypotheses will be verified without using statistical analysis as we will depend on our observations.

4.4 Results and Analysis

We monitor the change in behaviour between mIRC clients and *sdbot*. The results in Figure 4.2 show that it might be difficult to distinguish the normal behaviour from malicious behaviour because there is a noticeable change in log file size generated during a file transfer. We also notice that it is not sufficient to just look at network statistics. We can reject the first null hypothesis as it is sometimes difficult to distinguish between the normal and abnormal cases. Therefore, we need to use our correlation algorithm to provide a better clarification.

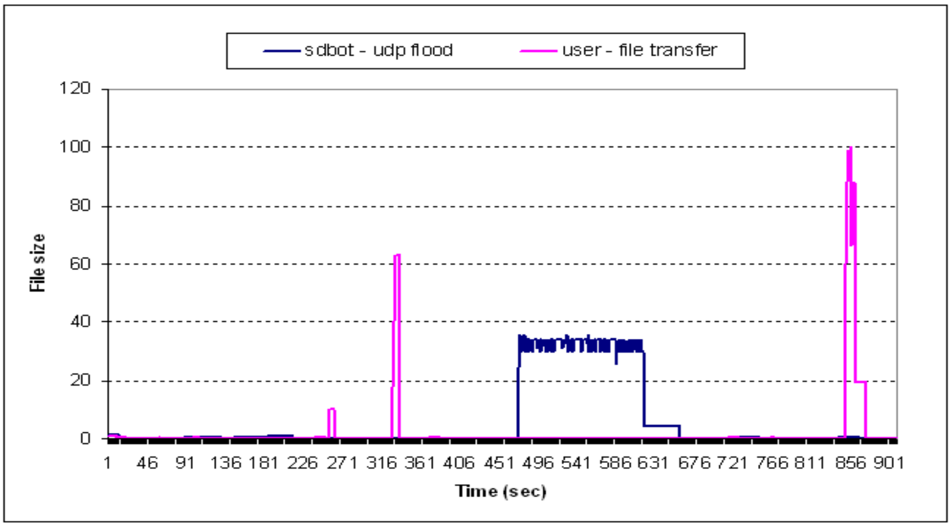

FIGURE 4.2: Change of log file size (a user transfers files vs. a bot using UDP flood. (100 bytes ≡ 275085 bytes).

4.4.1 Botnet Detection through Distributed Log Correlation

The results from the previous experiment show that sometimes it is difficult to distinguish the normal behaviour from malicious behaviour, e.g. when there is a sudden change in log file size. Therefore, we present our correlation detection scheme to distinguish between these two cases.

The basic idea is to find correlated events in different hosts. Since we are dealing with botnets, there is a high probability of having correlated events such as sending similar amounts of data to a bot herder that occurs within a specified time, or generating similar amounts of traffic to attack other systems. As a result, a high correlation between events is generated. A high correlation represents malicious activity, while a low correlation represents normal activity.

We have investigated the normal scenario of three users having normal conversations and using some IRC commands without transferring files to each other. The results show that there is a low correlation generated from the three users (Figure 4.3). We have also investigated the normal scenario of transferring files between users. The results show that even with a sudden change in log file size generated due to file transfer by user 3, we still notice a low correlation between data as shown in Figure 4.4. As a result, we can reject the second null hypothesis.

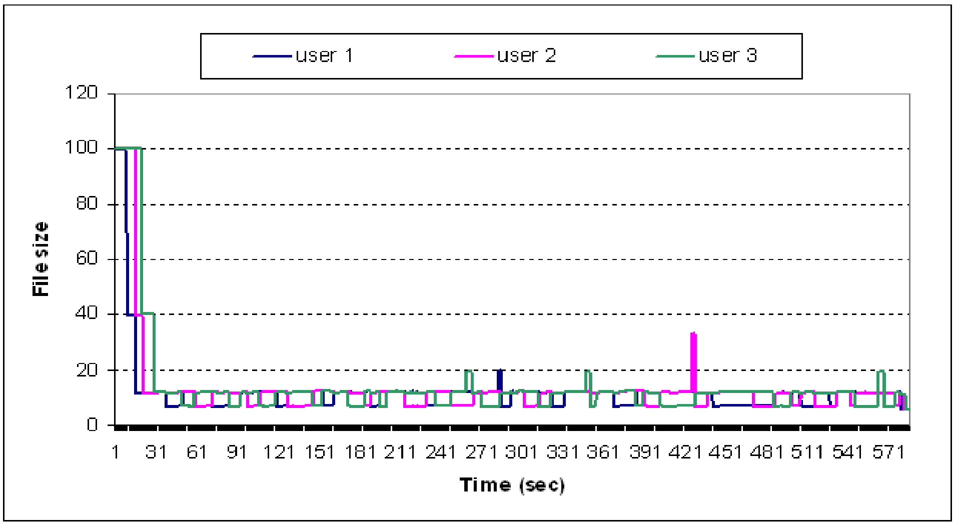

FIGURE 4.3: Normal users behaviour without sending files. (100 bytes ≡ 2754 bytes).

After simulating the normal cases, the three machines were infected by *sdbot*. This represents the two attack cases. In the first experiment, we have investigated the attack scenario of three bots receiving commands from their bot herder. No flood attack commands were received. We have noticed that the generated data is small but there is a high correlation between the changes in log file sizes as shown in Figure 4.5. In the second attack scenario, the bots receive flood commands from their herder. The results show that there is some obviously malicious activity in the network. This can be seen from the sudden change in the amount of data generated and the high correlation between the changes in log file sizes as shown in Figure 4.6. As a result, we failed to reject the third null hypothesis as the infected hosts generate the same amount of changes in file sizes when they receive data over the network.

The results from the correlation algorithm are shown in Figure 4.7. The x-axis

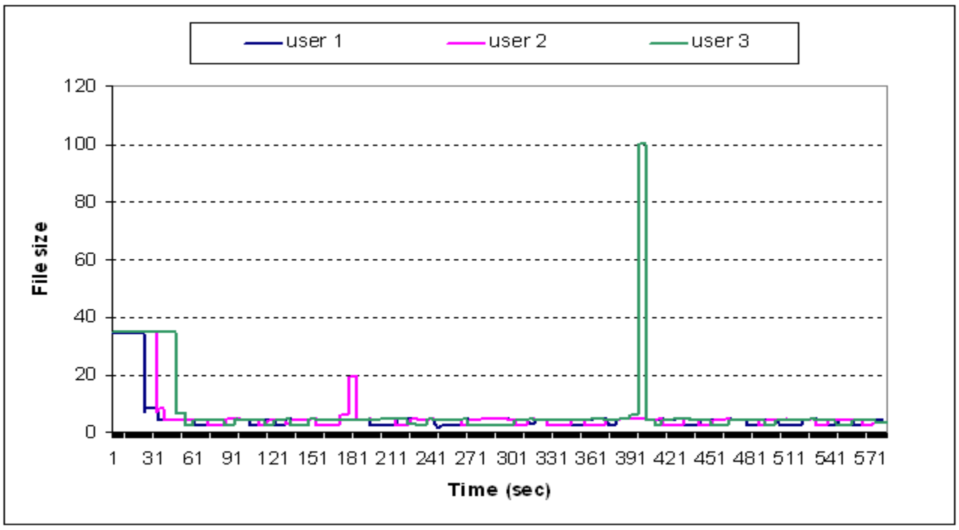

FIGURE 4.4: Normal users behaviour with sending files. (100 bytes ≡ 7248 bytes).

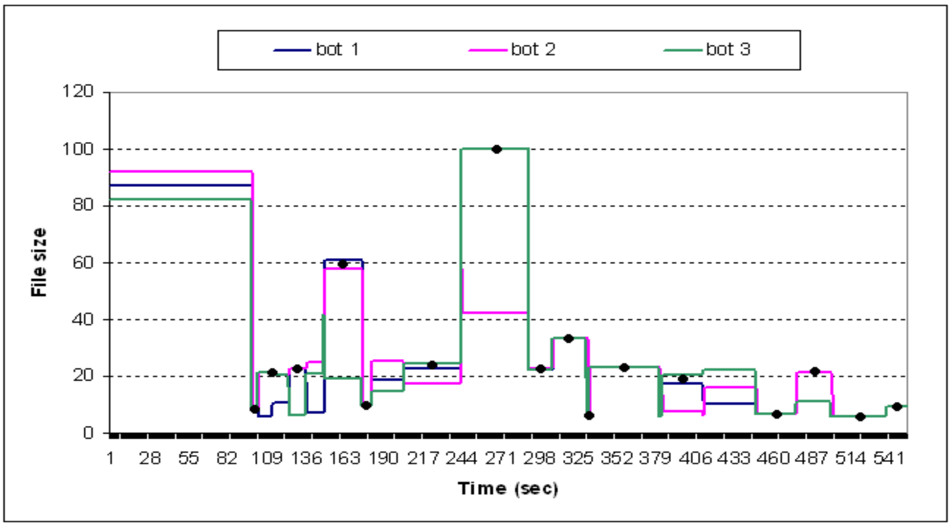

FIGURE 4.5: Attack behaviour without flood. (100 bytes ≡ 4121 bytes). The dots represents a high correlation between bots.

represents the normalized data while the y-axis represents the conducted experiments. We can see from the figure that we have a large number of uncorrelated events in the normal case. This represents a normal behaviour in our case since users are responding randomly to others. On the other hand, the uncorrelated events in the attack case are generated due to the fact that sometimes there is a delay in responding to the bot

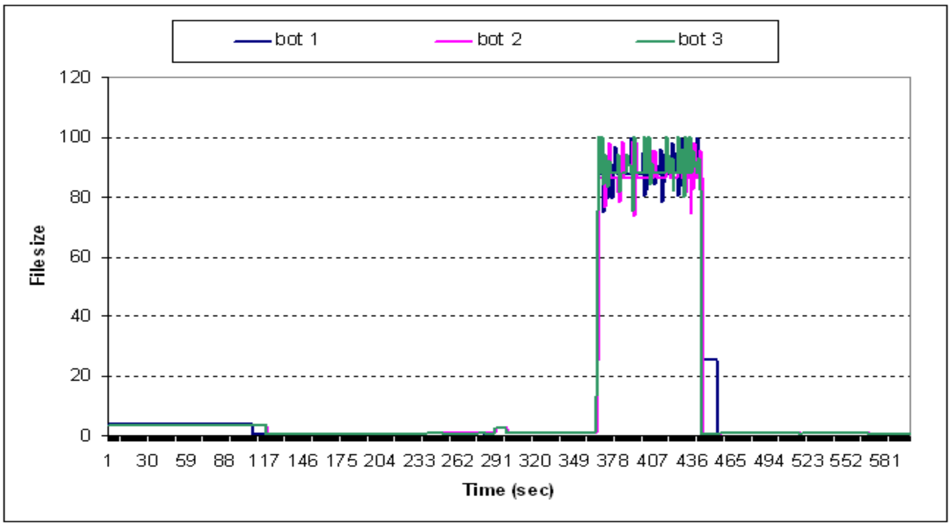

FIGURE 4.6: Attack behaviour with UDP flood. (100 bytes ≡ 94050 bytes).

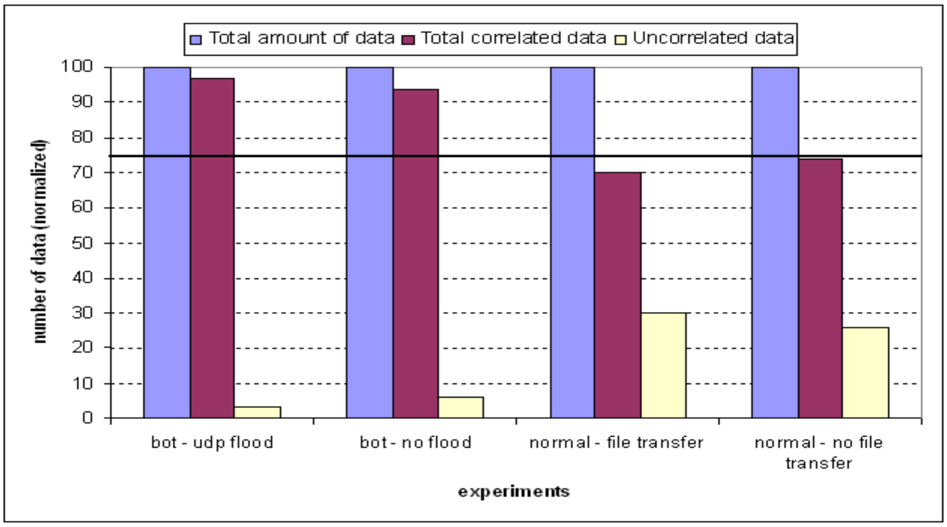

FIGURE 4.7: Correlation between log files.

herder's commands. We have also noticed that there are a large number of correlated events in the normal case. There are many reasons for this. The first reason is that we are running our experiments in virtual machines and switching between virtual machines takes some time. Another reason is that we are recording our data every second. Since, we have only one person (simulating to be three), recording data every second produces a large number of correlated events in the normal case.

To test how good our algorithm is in detecting botnets, we used a Receiver Operating Characteristic *(ROC)* analysis as shown in Figure 4.8. The x-axis represents cumulative false positive rates while the y-axis represents the cumulative true positive rates.

We set our threshold as a percentage of log file size. As we vary the threshold from 0% to 100%, we notice that our correlation algorithm detects abnormal activity when the value of the threshold is above 70% of the total amount of data and produces zero true negatives. Reducing our threshold to 70% generates one false positive (i.e. normal behaviour detected as attack). Setting the threshold below 70% generates two false positives while maintaining 100% detection rate.

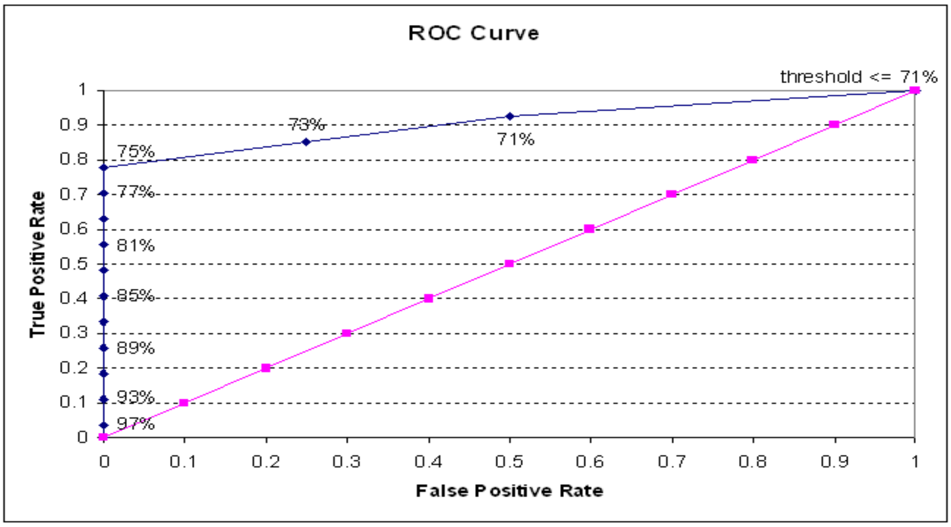

FIGURE 4.8: The ROC curve - false positive rate vs. true positive rate. The percentages represent the threshold used.

4.5 Discussion and Conclusion

Our results show that it is sometimes difficult to distinguish between normal behaviour and malicious behaviour. Therefore, we used an algorithm to detect bots based on change in behaviour by correlating events from different hosts. The correlation algorithm shows that there are a high number of correlated events in attack case generated by bots compared to normal users.

One problem with this approach is that we have to monitor a reasonable number of hosts to detect the correlation of similar events. In addition, these hosts should not be passive all the time. Otherwise, we will have a lot of zero correlation events which lead to large number of false positives.

One method to improve the detection technique of botnet is to consider not only correlation between events, but also to monitor the number and the frequency of executed *API* function calls.

Chapter 5

Host-Based Detection for IRC Bot using SRC Algorithm

5.1 Introduction

Our previous work was to detect the existence of botnet on the network by correlating data from different resources. Since little research has been done on detecting an individual bot on a machine [151][28], our main goal is to investigate this aspect. In the remaining chapters, our focus will be on detecting an individual bot running on the system rather than detecting botnet. There are many reasons for this choice. First, we think that detecting an individual bot on a system represents a much harder task than detecting botnets. This is because the bot can be very stealthy by generating small amount of traffic in different time period. Furthermore, the bot can use polymorphism and encryption techniques to avoid detection by different anti-viruses and intrusion detection softwares.

In addition to the common use of bots to generate a flood attack, an infected machine with a bot faces a problem of sensitive information leakage if the bot has the ability to intercept the user keystrokes through the keylogging process. Thus, if the user visits Internet sites which require the user name and the password, this information will be intercepted by a bot and sent to the attacker. In addition, the bot can steal the user sensitive information stored in his/her machine and use this information for malicious purposes. The bot can also manipulate the user files, and the attacker can remotely control the user machine through the existing bot.

In order to detect the running bot on the system, we have to look for common attributes or features an individual bot is using. Once we figure these attributes out, the next task is to investigate how we can correlate these attributes to enhance the detection mechanism. Another question is what sort of a correlation algorithm we have to use to achieve this task. To answer these questions, first we start to monitor the behaviour of the bots.

Attackers implement bots for different malicious purposes and tasks such as DDoS attack, spamming, or to store files. One of the main tasks of the current bots is to steal sensitive information from the user's machines such as credit card numbers, email passwords, social security numbers, bank account details or other personal information. This task is achieved by embedding a keystrokes logger as a part of the different functionalities of the bot when infecting the user's machines. Therefore, the first attribute that we have considered in our research is the availability of bot's keylogging activity.

Once the user's keystrokes are intercepted, the bot uses different ways to transfer them to the attacker. The first way is to store the intercepted keystrokes on a file and then read from that file. The other way is to store the intercepted keystrokes in a large buffer. Because storing the intercepted keystrokes process is needed for the bot as a part of sending the data to the attacker, we consider the reading from and the writing to a file as a second attribute that is to be taken into account. The final process is to communicate with the attacker. This is done by executing the communication function calls to transfer or receive the data to/from the attacker. Therefore, we consider monitoring the execution of communication function calls as our third and final attribute.

These three attributes may be performed simultaneously or within a short period of time. Therefore, correlating these attributes may indicate a malicious activity on the system. The difficulty that we have faced is how to correlate these attributes within a specified time period. In this chapter, we use a simple correlation method called a Spearman's rank correlation as a proof of concept.

Because our main focus is to detect bots by monitoring keylogging activity, we will present the keylogger concept and discuss how it is implemented and how it can be used to intercept the user's keystrokes. We will present the Spearman's rank correlation algorithm which correlates different events to detect an individual bot on the system with the results of the experiments conducted. Finally, we summarise and conclude our results and findings.

5.2 Keylogger Concept

A keylogger is a stealth program which records the user's activities such as user's keystrokes, screen logging, mouse logging, and voice logging. Keyloggers can be used for malicious activities such as capturing user names, passwords and credit card numbers. Most of the keyloggers are undetected by the anti-virus softwares because of their abilities to hide themselves from the task manager or processes list as they use different methods to log user activities. Keyloggers represent a serious threat to the privacy and security of the network. This threat is severe because keyloggers consume a small amount of memory making the detection process hard. In addition, many of the existing keyloggers are installed and executed without the need of administrator privileges. These keyloggers do not require loading of Dynamic Link Library (DLL) files. Furthermore, keyloggers store all captured keystrokes in a well-hidden file. The attacker can retrieve this file by allowing the keylogger to send it via email. The file can have the following options:

- send the log file when it reaches a certain size.
- clear the log after the log has been sent successfully.
- The format of log is either HTML or plain text.

The attacker can also retrieve this file by ftp server. Moreover, the attacker can encrypt the hidden file so that other software cannot decrypt this file.

There are two types of keylogger [155]:

1. Hardware Keylogger, and

2. Software Keylogger

A hardware keylogger is a device which is installed between the keyboard and the keyboard port on the motherboard. On the other hand, a software keylogger is a program that is installed on a PC in order to monitor the behaviour of a user. In this section, we will focus only on the software keylogger.

5.2.1 Keyloggers Forms

Most of the software keyloggers consist of two files. The first file is a DLL file in which a filter function is called. The second file is an EXE file which is used to load the DLL file and set the hook. Hooking is the way of intercepting an event before it reaches an application. Hooks provide powerful capabilities such as recording or playing back keyboard and mouse events. In addition, using hooking techniques can prevent another filter function from being called. Most of the keyloggers use a system-wide hook to filter messages of all applications.

5.2.2 Methods for Implementing Keystrokes Logger

There are three methods of implementing a keylogger. Two of these methods are implemented using two API function calls SetWindowsHookEx() and GetKeyState() in the user space while the third method is implemented in the kernel space (i.e. building a kernel driver).

The first method is through using SetWindowsHookEx(). SetWindowsHookEx() is a well known method for implementing a keylogger. SetWindowsHookEx() is an API function which allows the application to install a hook procedure to monitor events within the system. Whenever an event occurs in the system, the hook procedure is called. An attacker who is interested in building a keylogger uses WH_KEYBOARD to monitor keyboard strokes. To monitor all keyboard strokes, the hook procedure must reside in a DLL file. MSDN defines a hook [109] as: *"A hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure."*

MSDN also defines a SetWindowsHookEx() [111] as: *"The SetWindowsHookEx function installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread."* The function and the syntax for SetWindowsHookEx() is described below:

The syntax of SetWindowsHookEx() is defined below:

```
HHOOK SetWindowsHookEx(int idHook, HOOKPROC lpfn, HINSTANCE hMod,
DWORD dwThreadId);
```

In order to monitor keyboard events, idHook should set to WH_KEYBOARD. As we mentioned before, to use a system-wide hook and receive events globally, the hook procedure must be implemented in a DLL file. The steps of creating a hook procedure are shown below:

1. Create a DLL file with the hook procedure function which is used in SetWindowsHookEx() as shown below:

   ```
   LRESULT CALLBACK KeyboardProc(int code,WPARAM wParam,LPARAM lParam);
   ```

   The (wParam) parameter holds the virtual-key code (VK_*) of the key that generates the keystroke message and the (lParam) parameter holds other parameters such as the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag. These parameters are converted to ASCII characters and logged to a file every time a key is being pressed.

   MSDN states that the user should also check if the value of the first parameter (i.e. *code*) is less than zero. If the value of the first parameter (*code*) is less than zero, then the hook procedure must return the value returned by CallNextHookEx(). In addition, if the first parameter (*code*) is greater than or equal to zero, and the hook procedure did not process the message, the unprocessed messaged is also passed to CallNextHookEx() to allow other applications that use KEYBOARD hook to get notifications. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

2. The main application window will be created and hidden.

3. We implement the hook by using SetWindowsHookEx(), open a file for writing and wait for messages.

Another method of implementing a keylogger is by using a GetAsyncKeyState() function. GetAsyncKeyState() does not require a DLL file as in SetWindowsHookEx(). MSDN defines a GetAsyncKeyState() as a function which determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to GetAsyncKeyState(). To use this function, the user should define it as shown below:

```
SHORT GetAsyncKeyState(vKey);
```

The vkey specifies one of 256 possible virtual-key (VK_*) codes. To use this function, we need a thread, a loop or a timer that is constantly checked if a key is pressed and logged to a file.

Because the shift key state and caps lock state is unavailable, another API function GetKeyState() is used to handle this situation. GetKeyState() function allows the application to get the current state of the key if it is a letter or a digit (A through Z, a through z, 0 through 9). This method is based on getting keys from the foreground window. Another function GetKeyboardState() allows the user to retrieve the current status of all virtual keys by copying the status of 256 virtual keys to the specified buffer.

The third method to create a keylogger is by implementing a kernel device driver that monitors all I/O request packets that are sent to the keyboard driver. This method monitors all read requests and enables an I/O completion routine to be installed for each request. The key can be obtained whenever the read request completes making the I/O completion routine to be fired. The captured keys are either saved to a file or into a kernel space buffer in which applications which are located on user space can constantly obtain the keystrokes.

5.2.3 Existing Keylogger Detection Techniques

Detecting keylogger programs is a hard task because most of the keylogger programs are hidden. There are few existing keylogger detection techniques. One approach, used by Aslam [6] and his group, is to enumerate all the hidden and visible processes and DLL libraries and then disassemble each process and the related DLL. After disassembling the processes and DLL files, they search for SetWindowsHookEx() function used by the keylogger and highlight the suspicious items of each process. One problem with this method is that the keylogger developers can use different methods to log the user activities other than using SetWindowsHookEx(). In addition, disassembling all processes searching for SetWindowsHookEx() is a tedious task.

Research done on keylogging has shown that it is difficult to detect and the literature on this topic is also sparse. One method of detecting keyloggers is used by TAN [156] based on implementing a kernel-space device driver that hooks and disables some API functions such as SetWindowsHookEx(), GetKeyboardState(), and GetKeyState(). He stated that disabling these three functions prevents the proper functioning of keyloggers that utilise the first two keylogging techniques. However, disabling these functions will prevent legitimate softwares from using these functions. As part of this work, we monitor calls to some of these functions and other functions. Monitoring calls to these functions will not affect their use by legitimate programs.

Herley [63] uses a simple trick to hide the typed keys from the keylogger in a browser. He suggests that embedding a sequence of random characters between successive keys typed on the browser will make the keylogging process difficult. The keylogger will still receive the password but it will be embedded with random characters that make the password discovery infeasible. Although this technique is interesting, but it will be easy for the attacker to determine the correct keystrokes if the keylogger combines the keystroke and mouse click together. In addition, the keylogger may have ability to detect the currently working subwindow and thus intercepts the function call which produces these random keys.

Other methods implement an anti-keylogger software. The anti-keylogger software is divided into a signature-based anti-keylogger and a hook-based anti-keylogger. In signature-based anti-keyloggers, the detection is based on the applications that typically identify the keylogger based on the files or the DLL that is installed. The problem of this method is that it cannot detect unknown keyloggers which have different signatures. In hook-based anti-keyloggers, the detection is based on monitoring all processes that use the windows hook.

In addition to the existing keylogger detection techniques discussed previously, there are other suggested methods to defeat the keylogging techniques. Wang [170] detects the keylogger by first scanning the entire file system on the infected OS and clean OS and stores the results in infected and clean files respectively. Then, he uses WinDiff.exe program to compare the two and search for abnormal hidden files. Although this technique is very effective in detecting keyloggers on the system, it is difficult to implement for large scale networks because it requires every user interaction and cooperation which is a tedious task and is time consuming. In addition, this process can generate false positive because some benign programs may generate filenames that are too long or files could be created in a very short time period when the two scans are in process.

Another method implemented by Kanwar [85] to detect a keylogger is to monitor the GetAsyncKeyState API rate at which it is being called, or just find the executables which call this function. The SetWindowsHookEx() activity can be detected in the same way. One problem when using this technique is that this technique is based on calling (NtUserGetAysncKeyState, NtUserGetKeyState and NtUserGetKeyboardState) to read from keyboard, which can be blocked [169].

Shetty [134] suggests the use of a virtual keyboard when typing the passwords. Virtual keyboard is analogous to a graphical keypad where a user clicks on the characters rather than typing them on the keyboard. If the keylogger has a function to capture a screenshot, then this method will be defeated. This is because that the attacker can design his keylogger to take screen snapshot on every mouse click and then send this information to the attacker.

Because these keystrokes logging detection methods focus only on detecting the existence of the keylogger, we will present our methods for detecting bots which are based on monitoring the keylogging activities and sending the intercepted keystrokes logging to the attacker as part of their functionalities to steal sensitive information for the user's machines. In this chapter, we will present the Spearman's rank correlation algorithm which correlates different activities within a specified time period. In the next chapter, we will use the Dendritic Cell Algorithm (DCA) as a more intelligent correlation algorithm to correlate different bot's attributes and compare the results obtained with the Spearman's rank correlation algorithm. Finally, we will draw our conclusions for both correlation algorithms.

5.3 Spearman's Rank Correlation (SRC) Algorithm

5.3.1 Introduction

Existing research techniques detect the presence of bots via network monitoring. Rather than attempting to detect bots via network monitoring, our work focuses on a single bot detection on a machine by monitoring and correlating different activities on the system represented by executing different API function calls that may indicate the presence of a bot on the system.

The APITrace, an API monitoring program that we have developed, monitors and intercepts selected API function calls once they are executed by the bot in the *user-based* environment. Direct invocation kernel-based API functions will not be monitored by our intercepting program.

Our aim is to verify that correlating different behaviours of a single process within a specified time-window may indicate abnormal activity in our system. We use a Spearman's rank correlation algorithm in order to verify our aim. Our results show that correlating different attributes within a specified time period can enhance the detection algorithm.

This section is structured as follows: first, we briefly describe the methodology and the aim of these experiments. Secondly, we discuss the design and implementation of our experiments. Thirdly, we will present and analyse our results for different types of experiments that we have conducted and state the limitation of using Spearman's rank correlation algorithm. Finally, we will conclude and summarize our results.

5.3.2 Methodology

In this section, we try to detect the existence of the bot on the system by correlating different behaviours using APITrace tool. We focus on three types of bot behaviour: keylogging activity, file access and outgoing traffic. The APITrace monitors these activities by intercepting specified API function calls executed by the bot. Our main focus is to detect the keylogging activity on the system by monitoring *GetKeyboardState* or *GetAsyncKeyState* function calls. In addition, the APITrace intercepts other functions such as *WriteFile* and *ReadFile* for storing the intercepted data. Furthermore, the bot needs to send the intercepted data to the attacker, therefore, we may notice a large volume of outgoing traffic during this period. By tracking and correlating these kinds of behaviours, the process of bot detection will be enhanced.

In order to detect the existence of the bot in the system, different bot behaviours are correlated to have a high correlation value. We use the Spearman's Rank Correlation (*SRC*) value [8] to show how strong these activities are related.

The Spearman's rank correlation works as follow:

- Rank the two datasets
- Find the difference in the ranks, D
- Square the differences and sum them, D^2
- Calculate the coefficient R_s

The value of R_s is calculated from the following equation 5.1:

$$R_s = 1 - \frac{6 * \sum d^2}{n^3 - n} \tag{5.1}$$

So, the closer Rs to (+1) or (-1) the stronger the likely correlation.

An example of Spearman's rank correlation is given below: Suppose you have a set of data $A = 3, 2, 4, 2, 6$ and another set of data $B = 2, 2, 1, 3, 5$. In order to calculate the Spearman's rank coefficient R_s, we follow the process described earlier as show in Table 5.1.

TABLE 5.1: Spearman's Rank Correlation Example

Set(A)	Set(B)	Set(A^*)	Set(B^*)	Rank(A*)	Rank(B*)	D	D^2
3	2	2	2	1.5	2.5	-1	1
2	2	2	3	1.5	4	-2.5	6.25
4	1	3	2	3	2.5	0.5	0.25
2	3	4	1	4	1	3	9
6	5	6	5	5	5	0	0

Therefore, using equation 5.1, we can fine the value of $R_s = 0.175$, which shows a low correlation between set(A) and set(B).

A threshold of ±0.5 is used to test the correlation strength between two activities. If the Spearman's Rank Correlation value exceeds the threshold level of ±0.5, a high correlation between the two activities is observed which may reflect malicious activity on the system. According to the Spearman's Rank Correlation algorithm, the level of ±0.5 or more represents a strong correlation between two events. We hypothesise that one behaviour may not be enough to detect malicious activity. For example, one behaviour of the bot is to send the intercepted information from the keylogging process to the botmaster, once the botmaster issues the keylogging command. The intercepted information is sent to the botmaster if the user of the infected machine hits the [ENTER] key, or closes the active window. This action represents normal behaviour. Correlating different actions enhances the process of detection. We are aware that different keyloggers use different techniques to intercept and store the keystrokes. Our work examines a keylogging activity as our goal is to detect a bot rather than a keylogger.

5.3.3 Aims

The aim of our experiments is to verify the notion that correlating different behaviours of a single process which produces multiple API function calls within a specified time-window, indicates abnormal activity. In addition, we apply the monitoring and correlation scheme to a normal application (e.g. mIRC client) to verify that the normal application executes different function calls from the malicious process which results in having different correlation value.

5.3.4 Design and Implementation

In our research, we focus on monitoring selected API function calls executed by bots that perform the keylogging task and send the intercepted keystrokes directly to the IRC channel. To accomplish this task, we use the APITrace to monitor the selected API function calls executed by the bot when receiving commands from the botmaster. We focus on three types of API function calls: (1) the communication functions, (2) the file access functions and (3) the keyboard status functions as explained in Section 3.6.

In this work, we captured selected API functions such as GetKeyboardState, and GetAsyncKeyState used by bots which implement keylogging feature. For example, the spybot [12] is used for its ability to intercept the user's keystrokes by invoking GetAsyncKeyState. We search for all the running processes in our system and use APITrace tool to inject our hooking program into the running processes. An API hook is based on modifying the process Import Address Table (IAT) [2] to point to the replacement function instead of the original function. Thus, we were able to capture the functions made by the bot when it receives commands from the botmaster.

We store the captured functions in a log file for further processing. We use a Spearman's Rank Correlation formula [8], explained previously, to find the correlation between different behaviours of the bot. As we are trying to detect the keylogging activities in the host (presented as S_1 in 2) and correlate it with other activities such as sending the intercepted keystrokes directly to the IRC channel within specified time-window (S_3) to enhance the performance of bots detection, we use the keylogging activity as a primary set. In addition, we also correlate the events of intercepting the user keystrokes (S_1)and file access or registry access (S_2). This kind of signal selection is based on causality, sequences of events happened in response to each other. Once the bot intercepts user's keystrokes or reads files from user machine, it needs to send it to some place which is most propably the attacker. This kind of process leads us to the right selection of behaviours. If a wrong order of behaviour selection is chosen it can lead to false alarms. The SRC algorithm for detecting the bot is described in Algorithm 2.

```
$S_1$: keystrokes interception (KeyboardState functions);
$S_2$: File/Registry Access Activities (FileAccess);
$S_3$: Communication functions (CommFunc);
if KeyboardState function(s) is executed (i.e. keylogging activity) then
    if SRC($S_1$,$S_3$) > Threshold and SRC($S_1$,$S_2$) > Threshold then
        Strong detection;
    else if SRC($S_1$,$S_3$) < Threshold and SRC($S_1$,$S_2$) < Threshold then
        Weak detection;
    else if [SRC($S_1$,$S_3$) < Threshold and SRC($S_1$,$S_2$) > Threshold] or [SRC($S_1$,$S_3$) >
    Threshold and SRC($S_1$,$S_2$) < Threshold] then
        Medium detection;

else
    No detection and normal activity is considered;
end
```

Algorithm 2: Bot Detection Algorithm using Spearman's Rank Correlation (SRC)

We have performed five experiments to verify our notion. In the first experiment (*E1*), we allow the bot (spybot is used because it has keylogging capabilities) to connect to the IRC server and join the channel without receiving any commands from the botmaster. In the second experiment (*E2*), we follow the same procedure as in the first experiment, but in this case the botmaster issues different commands to the bot, excluding the keylogging activity. Note that our target machine in these experiments is an idle infected machine. That is, the user does not use the infected machine for any activity.

In the third experiment, we allow the bot to connect to IRC server and join the specified channel. The bot on the infected machine monitors the user's typing activity, but does not send any information to the botmaster. We monitor two scenarios of typing. In the first scenario (*E3.1*), the user types long sentences while in the second scenario (*E3.2*), the user types short sentences. By monitoring two typing scenarios, we are able to show the effect of different user's activity on our detection scheme.

In the fourth experiment, once the bot connects to the IRC server and joins the channel, the botmaster starts the keylogging activity. The same procedure is followed as in the third experiment where we have two scenarios of typing: long sentences (*E4.1*) and short sentences (*E4.2*).

The fifth and the final experiment (*E5*) involves applying the monitoring program to another application (*mIRC* client [106]) to verify that *mIRC* client behaves differently from the bot.

Each experiment is performed five times which is sufficient as the results from the repeated experiments produce only small variations by using Chebyshev's Inequality due to network delay and through using VMWare. Therefore, we selected a random experiment from the repeated experiments as the base experiment. Each experiment runs for 15 minutes in order to collect a reasonable number of function calls, which cover most of the botmaster execution commands. The monitored API functions are saved into a log file. After that, we use a Spearman's Rank Correlation (SRC) method to correlate different behaviour of the bot based on the frequency of API function calls executed by the bot in our system within a specified time-window. In our experiments, a time-window of ten seconds is used between function calls samples. We have noticed that monitoring function calls for a time-window of 60 seconds will have variant idle periods depending on the bot activity. An idle period is where no bot activity is detected and zero values are assigned. Therefore, using a time-window of ten seconds reduces the idle periods suitably.

The Spearman's Rank Correlation correlates two different datasets. The first dataset is the outgoing traffic from our system (i.e., total number of bytes sent to the botmaster every ten seconds) and the frequency of GetAsyncKeyState function calls generated. The second dataset is the frequency of GetAsyncKeyState function calls and the frequency of WriteFile function calls generated. These function calls are important for monitoring bot behaviour because their invocation represents abnormal behaviour within our system.

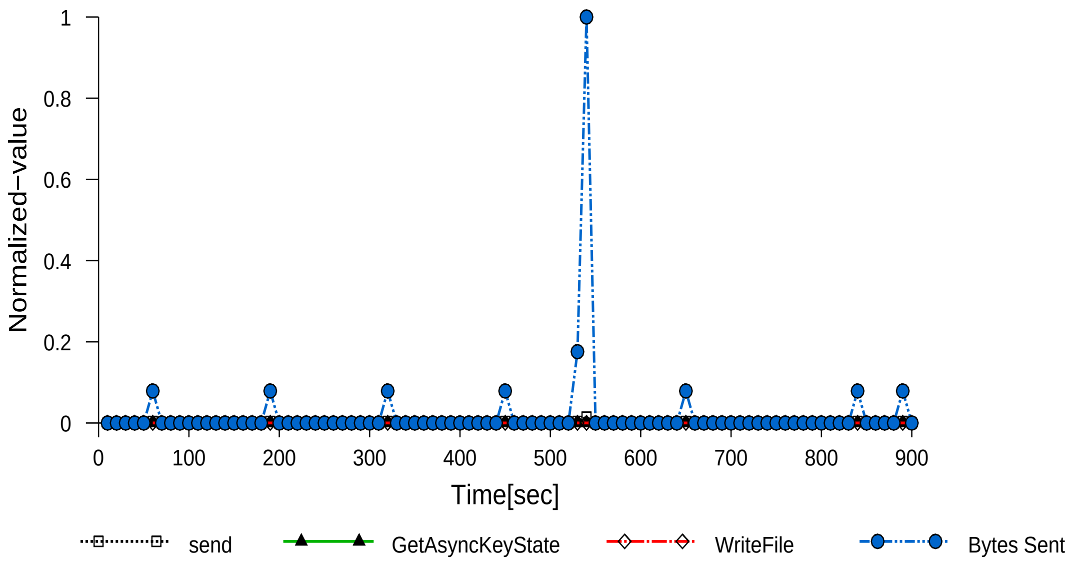

FIGURE 5.1: The results of experiment E1. The bot connects to the IRC server, joins the specified channel and remains inactive waiting for the botmaster's commands.

5.3.5 Results and Analysis

In this section, we analyse the results of the experiments described in Section 5.3.4. For all experiments, the x-axis represents time in seconds while the y-axis represents the normalized value of functions. The normalized function frequency call values represent the total value we get during 10 seconds divided by the maximum value of the whole period (900 seconds). In addition, we use a line graph which connects the points to make our figures more readable.

In the experiment *E1*, the bot is idle for a major part of the duration. This means that no API function calls are executed except the communication functions, specifically, *send* and *recv*, as shown in Figure 5.1. From Figure 5.1, we notice that it is difficult to detect the bot's behaviour as there is no activity in the system except the communications. We also notice that there is a burst in the outgoing traffic. This burst is generated due to spybot program which sends a bulk of words at every specified time interval.

In the experiment *E2*, the botmaster issues commands such as *info*, *list* and *pass-*

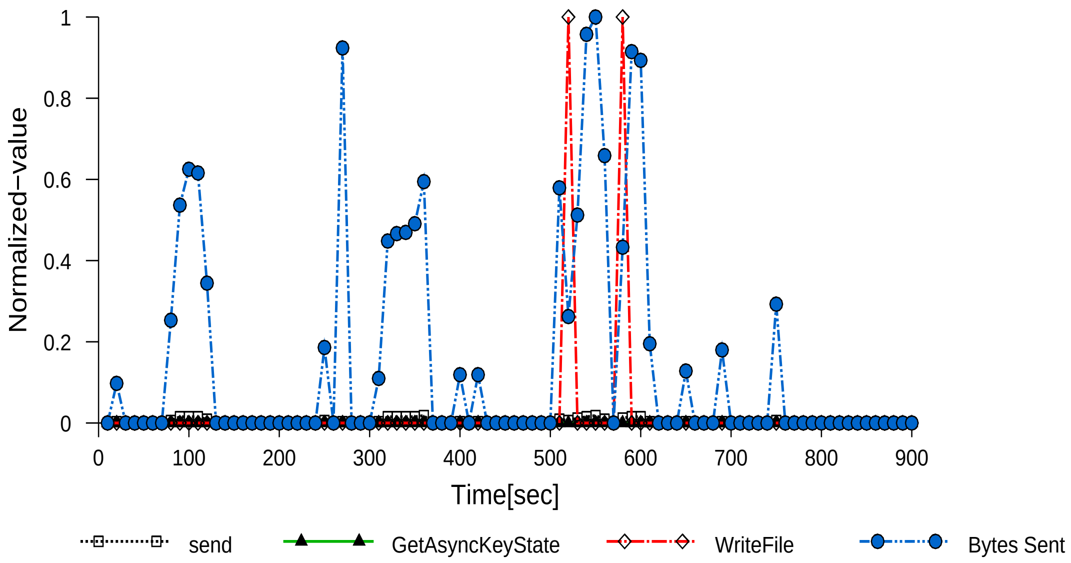

FIGURE 5.2: The results of experiment E2. The bot receives commands from the botmaster. The amount of outgoing traffic increases as the bot responds to the botmaster's commands.

words and the bot on the infected machine responds to these commands. Each time the botmaster issues a command, different API function calls are executed by the bot. In this experiment, we noticed an increased amount of outgoing traffic compared to the experiment *E1*. In addition, few WriteFile and ReadFile functions are generated during this experiment. Conversely, no GetAsyncKeyState function calls are generated, as shown in Figure 5.2.

The third experiment has two typing scenarios: (1) Long sentences (*E3.1*) and (2) Short sentences (*E3.2*). Figure 5.3 represents the long sentences scenario *E3.1*. We notice that even though we have many GetAsyncKeyState function calls executed by the bot, which indicates keylogging activity, there is almost no correlation between GetAsyncKeyState and WriteFile. This is because the WriteFile function call is rarely generated as it is only triggered when the user types long sentences. To save the long sentences, the user has to press the [Enter] key or close the application. In addition, no data is sent to the botmaster which reduces the correlation value between GetAsyncKeyState and the outgoing traffic. In scenario *E3.2*, the user of the

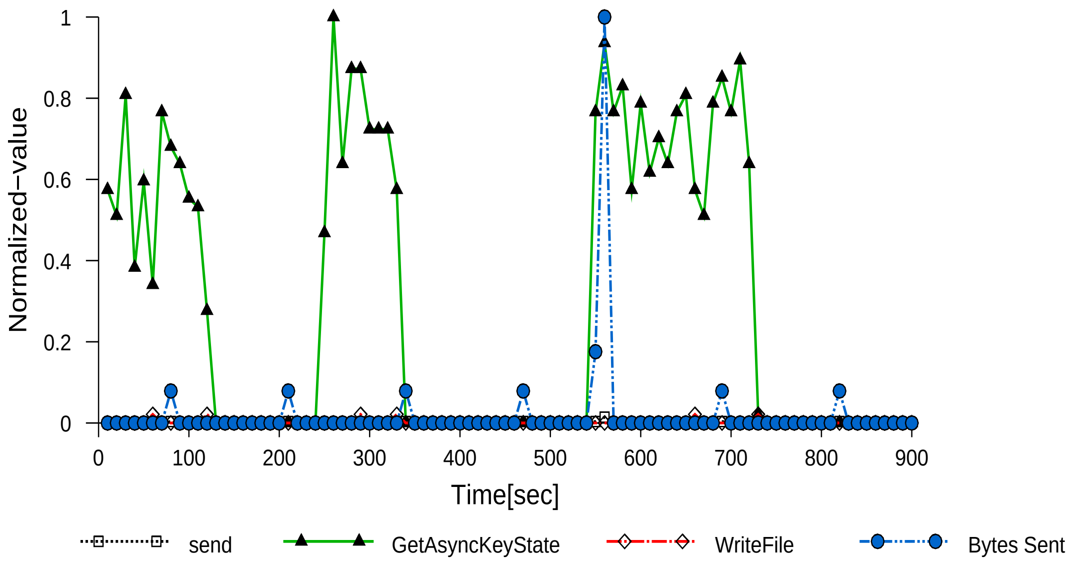

FIGURE 5.3: The results from the third experiment - scenario E3.1. The botmaster has not activated the keylogger command. The user on the infected machine types long sentences.

infected machine types short sentences. We can see from Figure 5.4 that there is a high correlation between GetAsyncKeyState and WriteFile function calls. This situation is expected as each time the user types short sentences, the functions GetAsyncKeyState and WriteFile are called to intercept the user keystrokes and store them in a file. However, there is still no traffic sent out and hence there is no correlation with outgoing traffic.

In the fourth experiment, the botmaster starts the keylogging activity and the intercepted keystrokes are sent to the botmaster. In this case, we also have two typing scenarios: (1) Long sentences (*E4.1*) and (2) Short sentences (*E4.2*). In scenario *E4.1*, we expect there to be a high correlation between the outgoing traffic and GetAsyncKeyState. However, the result from Figure 5.5 shows that there is a low correlation between the two. This is because we correlate the two events (typing and saving to a file) in two different 10 second time intervals. In addition, the long sentences increase the idle time, and therefore reduce the correlation value. Moreover, a low correlation between GetAsyncKeyState and WriteFile is noticed. This situation

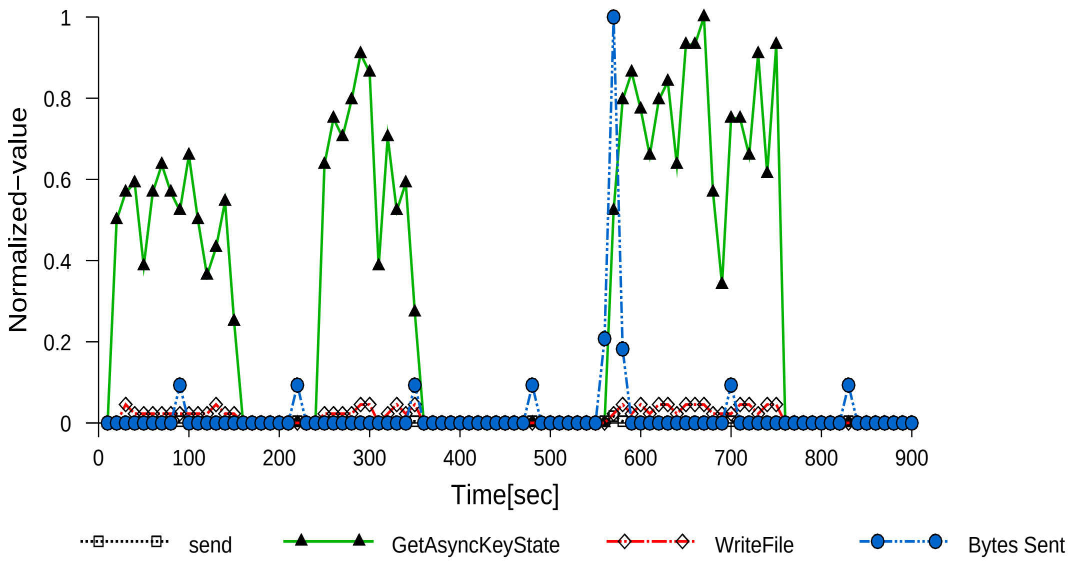

FIGURE 5.4: The results from the third experiment - scenario E3.2. The botmaster has not activated the keylogger command. The user on the infected machine types short sentences.

is expected as the user types long sentences which call few WriteFile functions.

In the second scenario *E4.2*, the user types short sentences resulting in a high correlation between the outgoing traffic with the GetAsyncKeyState function and between the GetAsyncKeyState function and the WriteFile function as shown in Figure 5.6. The high correlation in both cases increases the amount of evidence for a bot spying on our system.

In addition, we test our monitoring program with the mIRC program. The result in Figure 5.7 is optimistic as the program did not call any GetAsyncKeyState or GetKeyboardState functions.

Table 5.2 represents the value of Spearman's Rank Correlation between the two datasets, (`freq(GetAsyncKeyState), Bytes Sent`) and (`freq(GetAsyncKeyState), freq(WriteFile)`), in each experiment. In this table, we have two sets of results. In the first set *S1*, we correlate all the captured data from our algorithm including the idle period. In this period, no activity is seen, therefore, we assign a zero value to this period. This is represented by the *with zero* column in Table 5.2. In the second

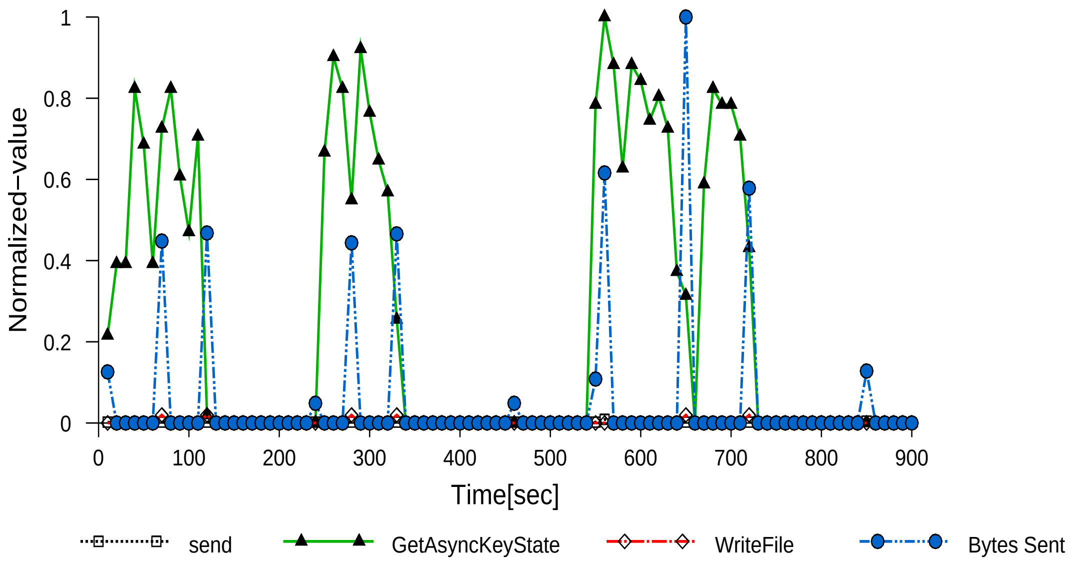

FIGURE 5.5: The first scenario E4.1 in experiment four. The botmaster activates the keylogger. The user on the infected machine types long sentences.

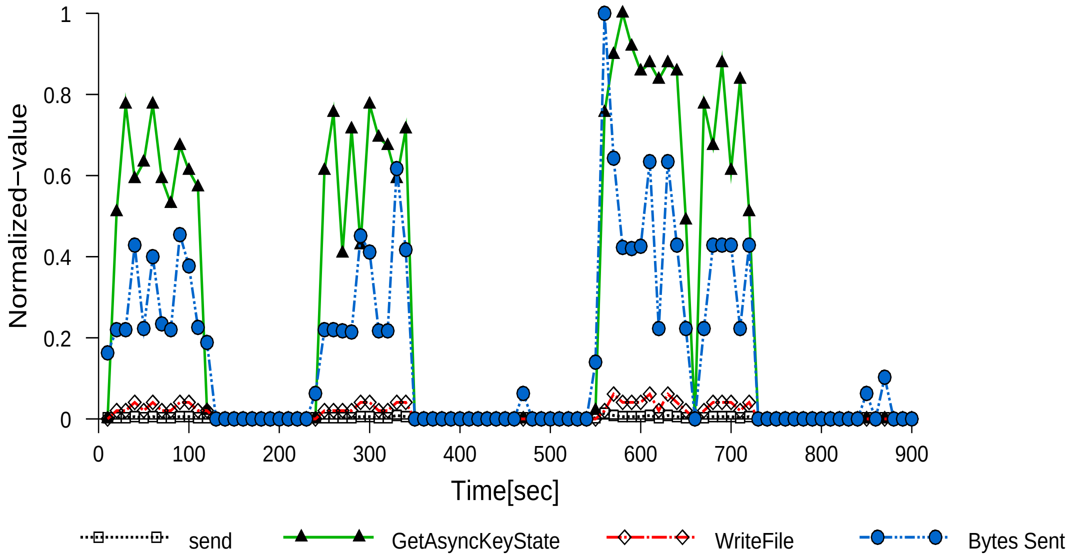

FIGURE 5.6: The second scenario E4.2 in experiment four. The botmaster activates the keylogger. The user on the infected machine types short sentences.

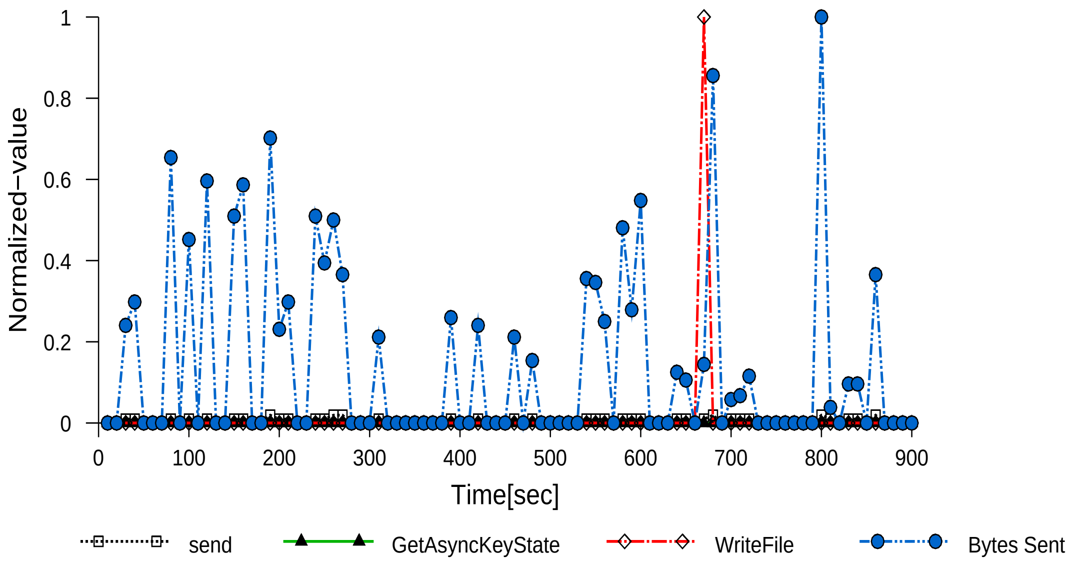

FIGURE 5.7: The results from Experiment E5. The mIRC client connects to the IRC server. The client has normal conversation and simple commands with another client.

set *S2*, we remove all the idle periods which have zeros and apply the Spearman's Rank Correlation to the new data. The reason for having the two sets is that we have noticed that having the idle periods in our data increases the correlation value. This is because there are many places where no activity is noticed in both datasets, which may produce inaccurate correlation. Therefore, we wanted to investigate the effect of having no idle periods. Although we have noticed a reduction in the correlation value by 0.35 in most cases when we remove the idle periods, it gives us more accurate results.

The *API Keylogging Activity* column represents the situation where the process calls any function used to intercept the keystrokes such as GetAsyncKeyState, GetKeyboardState, GetKeyNameText and keybd_event. Calling these functions may indicate a keylogger activity. As a result, we classify our detection scheme into four cases:

- No detection (N/A): the case where no keylogging activity is detected.
- Weak detection (Weak): the case where a keylogging activity is detected but a

low correlation is noticed in both datasets.

- Medium detection (Medium): the case where a keylogging activity is detected but a high correlation is noticed in one dataset.
- Strong detection (Strong): the case where a keylogging activity is detected but a high correlation is noticed in both datasets.

As mentioned in section 5.3.2, a high correlation is considered if the Spreaman's Rank Correlation value exceeds the threshold (0.5). Conversely, a low correlation value is considered if the Spearman's Rank Correlation value is below the threshold.

From Table 5.2, we see a perfect correlation of GetAsyncKeyState and WriteFile function calls in the experiment *E1*. The bot called neither of these functions during its inactive period. We have also noticed that there is a high Spearman's Rank Correlation value between the outgoing traffic (Bytes Sent) and GetAsyncKeyState because the amount of outgoing traffic is equal each time. This traffic belongs to the PONG message generated by the bot to avoid disconnection from the IRC server. Therefore, the correlation value is expected to be high as well. In the experiment *E2*, the high Spearman's Rank Correlation value is due to the correlation of GetAsyncKeyState and WriteFile which are not invoked and zero values are assigned.

In the experiment *E3.1*, we have noticed a call to GetAsyncKeyState which indicates abnormal activity. On the other hand, a low Spearman's Rank Correlation value is generated in both datasets. This situation is expected because the user types long sentences which make only a few calls to WriteFile and no information is sent to the botmaster. As a result, a weak detection is indicated. Experiment *E3.2* detects a keylogging activity and generates a high correlation between GetAsyncKeyState and WriteFile executed by the bot due to typing short sentences. On the other hand, no information is sent to the botmaster which results in medium detection according to our classification.

Experiment *E4.1* shows similar activity to experiment *E3.1* where the user types long sentences, but the information is sent to the botmaster. We expect to have a high correlation between the outgoing traffic and GetAsyncKeyState function. The result

TABLE 5.2: Spearman's Rank Correlation (SRC) value which represents the correlation between two datasets.

Experiments	SRC(GetAsyncKey, Bytes Sent)		SRC(GetAsyncKey, WriteFile)		Keylog. Activity existence	API Detection confidence
	with zeros (S1)	without zeros (S2)	with zeros (S1)	without zeros (S2)		
E1	0.863	0.671	1.000	1.000	No	N/A
E2	0.648	0.498	0.967	0.897	No	N/A
E3.1	0.509	0.183	0.559	0.172	Yes	Weak
E3.2	0.423	-0.003	0.928	**0.618**	Yes	Medium
E4.1	0.506	0.189	0.560	0.089	Yes	Weak
E4.2	0.927	**0.579**	0.957	**0.663**	Yes	Strong
E5	0.594	0.499	0.983	0.958	No	N/A

shows there is no significant difference from experiment *E3.1*. This is because the bot sends the information when the user finishes typing long sentences. Experiment *E4.2* is the best case for detecting keylogging activity in our system. In this experiment, we detect the keylogging activity and we have high correlation values for both datasets which indicate abnormal activity running in our network.

The last experiment *E5* in Table 5.2 shows the result of the Spearman's Rank Correlation on monitoring the mIRC client. Even though we have a high correlation value before and after removing idle periods on both experiments, we did not detect the use of keylogging function calls. The high correlation value between *outgoing traffic* and GetAsyncKeyState relates to the number of idle periods due to the delay in responding to other client's messages.

In summary, we have noticed that some experiments produce high correlation values. There are many reasons for this. The first reason is that different events occur in different time-windows. Therefore, our algorithm produces inaccurate results. The second reason is that we have many idle periods in our datasets. The idle periods increase the correlation value which affects our detection scheme. In order to improve our detection scheme, we need to apply a more intelligent correlation scheme.

5.3.6 Limitation of the Algorithm

From our results, we have showed that the SRC algorithm can detect a bot based on correlating different behaviours within specified time window. A higher correlation value increases a confidence of having malicious activities on the system. Because the SRC algorithm correlates activities within specified time period, it is possible for the botmaster to evade this detection technique by allowing the bot to wait for a random time before performing another task. In this situation, we need to increase the time window and search for correlation of events within this time window. Increasing the time window may have a negative impact on our bots detection period.

Another important issue is that we focus on detecting a bot based on its keylogging activity. Combining other bot activities such as SYN attack or UDP attack with the keylogging activities can increase the correlation between different events.

5.3.7 Summary

In this section, we use the APITrace program to monitor selected API function calls of the spybot. We consider the execution of these functions within specified time-window as a security risk to our system. We highlight that looking at API function calls alone is not sufficient as other normal applications can call the same functions. As a result, the need for correlating different behaviour data is recommended. We use a Spearman's Rank Correlation method to correlate our captured data. Although the Spearman's Rank Correlation is a simple method to examine the correlation level, the results were promising. In addition, our results show that including idle periods in our correlation algorithm produces inaccurate results. This is due to the fact that most of the situations examined had a large number of idle periods which increases the correlation value. A more intelligent way of correlating the data is to remove the idle periods. Although removing the idle periods reduces the correlation value, it produces more acceptable results. It would be even better to remove only certain idle periods, something we will consider in our future research.

Other experiments show a weak detection decision (low correlation values). This is because different activities occur in different time-windows. For example, the typing process has a different time window than writing to a file or sending information to the attacker. We believe that choosing a correct time-window as well as the window size to correlate our data will have a large effect on our detection algorithm. In the next chapter, we will use the Artificial Immune System (AIS) approach to correlate different activities within the same time window.

Chapter 6

Host-Based Detection for IRC Bots using Dendritic Cell Algorithm (DCA)

The exchange of concepts from biology to computer science has been an interesting area for many researchers using biological inspired computation. For example, artificial immune systems (AIS) are theoretical biological inspired algorithms based on the functions, principles and modules of human immune system [23]. Artificial immune systems (AIS) are applied to complex problem domains such as computer security. This is obviously shown by developing different algorithms which try to mimic the human immune system to detect or prevent virus or worm attacks against computers and networks. The first generation of AIS represents a simplified version of immune models. The new generation of AIS tries to understand the human immune system functionalities. These algorithms show optimistic results when applied to different applications such as anomaly detection, pattern recognition and robotics.

6.1 Introduction

Human immune system plays a vital role in defending the human body against different types of attacks from pathogens such as viruses and bacterias. This is because the human immune system is robust, distributed and error-tolerant. Because of these features, the human immune system provides protection for the body from invading microorganisms termed pathogens. The protection process involves rapid detection, removal and repair of the infection. For these reasons, computer scientists try to some extent mimic the behaviour and the functionality of a human immune system in the area of computer security (i.e. intrusion detection or pattern recognition) to detect different attacks from viruses, worms or trojans. The human immune system functionality is applied to develop Artificial Immune Systems (AIS) [23] and feasible algorithms are developed mimicking the concepts of immunology. Most of the current work in AIS is on the first generation of AIS such as negative selection, clonal selection and immune networks. Unfortunately, these algorithms suffer from different limitations such as generating large number of false alarms or suffer from scalability issue. In order to solve these problems, a second generation of AIS is presented. One of these second generation AIS algorithms is called the Dendritic Cell Algorithm (DCA) which is developed using a multi-scale approach. The DCA is based on the abstract model of dendritic cells (DCs). The input data is in the form of two input streams which are combined and correlated across variable windows.

This section is structured as follows: We briefly present some background about the human immune system in section 6.2. Then we move on to the artificial immune systems and discuss different algorithms that have been implemented such as Clonal selection and Negative selection algorithms in section 6.3. After that, in section 6.4, we briefly explain the danger theory approaches focusing on and explaining the Dendritic Cell Algorithm (DCA). We introduce the bot scenarios and how we collected and presented our data in section 6.5. Section 6.6 is related to the experiments that we have conducted in which we state different null hypotheses that are used for this investigation. We present and discuss our results for both Spearman's rank correlation algorithm and the DCA in section 6.7. We conclude with a discussion of results in section 6.8.

6.2 Human Immune System

Human immune system consists of multiple cells to protect the human body from invading pathogens. It is subdivided into two distinct classes: the innate and the adaptive systems. The innate immune system was first observed by Metchikoff in

1882 [135] which represents the first line of defense against pathogens in a human body [48][55]. It provides a fast reaction and performs initial pathogen detection within a body by instructing the immune system to take actions against the invasion. The immune cells have receptors to sense molecules which indicate the presence of pathogen. The immune system consists of population of cells which are able to restructure their receptors to adapt to new threats. Such cells in the adaptive immune system have the ability to reorganize the molecules of their pathogen detectors and attempt to adapt to new threats. These cells are termed B-cells and T-cells. The combination of fast response from the innate immune system and the dynamic modification of adaptive immune system provides sufficient protection to the human body against different pathogen.

Previously, theories in the field of immunology rotated around the ability of the immune system to distinguish between ‘self’ (normal) and ’non-self’ (abnormal) entities. In 1891, Paul Ehrlich [136] showed that antibodies are the defense mechanism against pathogen termed antigen. They suggested that these antigen should be removed by the immune system before an infection without causing damage to its own cells and the body. After his work, it was discovered that a particular white blood cell termed B-cell is responsible for the production of antibodies [48][55].

The ‘self-nonself’ theory raised many questions in the field of immunology. In order to modify the classical ‘self-nonself’ viewpoint, two major theories were proposed namely the infectious nonself principle and the danger theory. The infectious nonself principle states that in addition to the binding of antigen to complimentary T-cell a second signal (termed costimulatory signal) is required in order to activate the immune system [101][55][47].

After the characterization of antibodies, Burnet [135] introduces the clonal selection theory. He found that ”an individual somehow manages to prevent all future ability to respond to auto antigen (self), leaving intact to respond actively to the universe of other antigens (nonself)” [55].

In 1959, the principle of negative selection as a classical one-signal model was presented by Joshua Lederbeg and a second white blood cell is categorized and termed T-cell [48]. He suggested that “whenever produced, T-cells undergo a period of immaturity during which antigen recognition results in their deaths" [27]. He states that in order for cell to remove pathogens, a further activation of T-cell is needed in the tissue [55].

The theory that is introduced by Ehrlich has a problem of self cell being attacked by the antibodies. In order to solve this problem, a costimulation concept is introduced and the idea of a B-cell working in conjunction with a T-cell is adopted forming a two-signal model [135]. In order for a T-cell to become activated, it must be presented antigen by an antigen presenting cell (APC) in conjunction with co-stimulatory molecules (CSM) [47]. It is also suggested that the B-cell would be eliminated if it did not receive a costimulatory signal from a 'helped T-cell' which is regulated by a 'stimulator cell' that provides the costimulatory signals. These cells are known as dendritic cells (DCs) and part of the innate immune system. A two-signal model was first proposed by Hofmeyr [66] and implemented by Balthrop [7] where it was shown to reduce false positive rates in different cases [48].

Another improvement to the constimulation process is introduced by Janeway [81][48]. In his work, he suggested that the DCs can perform their own version of self-nonself discrimination by recognizing the signature of bacterial presence innately. Note that the DCs can be altered for binding to molecules produced by bacterias. These molecules are termed pathogen-associated molecular patterns (PAMPs) [47]. Whenever a DC is exposed to PAMP and antigen, it produces a collection of molecules which increases the time a T-cell remains in contact with a presented antigen which assists in the activation of T-cells.

Until the early 1990s, some questions were still unanswered. It was not clear why the immune system should respond to self and why PAMPs which are produced by some bacterias are not classified as foreign. In 1994 Matzinger proposed a third-signal model. He stated that the immune system responds to the detection of damage to the body not to the detection of specific antigen structures [102]. In addition, he suggested that such activating danger signals are generated when a cell dies unexpectedly (necrosis). Furthermore, the danger theory also states that the immune system can be suppressed by a safe signal in the absence of danger or when cell dies normally (apoptosis) [48][55][47].

PAMPs, danger signals and safe signals have impact on DCs. A high concentration of PAMPs and danger signals transform the DC to a 'mature state' in which a T-cell becomes activated and all entities bearing that antigen are removed. In contrast, a high concentration of safe signals leads to a 'semi-mature state' in which the T-cell is tolerised to the presented antigen and no response is generated. This process suggests that the foreign object is harmless, and then the immune system does not respond to it.

6.3 Artificial Immune Systems (AISs)

AIS are systems and algorithms inspired by the functions of human immune system. It started with self-nonself principles of negative selection to create the negative selection algorithm used for computer security applications and clonal selection to clonal selection algorithm used in different areas of immune algorithms.

The difference between Artificial Immune Systems (AIS) and genetic algorithms is that in genetic algorithms there is no single algorithm from which all immune systems are derived. In contrast, the research in AIS is parallel, integrated and adaptive. The first generation of AIS is based on a simple module of immunology which reflects the initial inspiration such as negative selection and clonal selection while the second generation such as Dendritic Cell Algorithm (DCA) [54] has a complex model which tries to understand the behaviour of immune system in greater detail.

In the next sections, we will describe the first generation of AIS and then move towards the second generation of AIS, which we use it for bots detection.

6.3.1 Clonal Selection

Researchers discovered that the B-cell is responsible for producing antibodies which are stimulated upon encountering a foreign antigen. The resulting B-cell clones vary the receptor configuration to perform a local search to find the best fitting receptor. Clonal selection was initially based on works by Burnett in 1970s [20]. In his work, he characterizes the affinity metrics using mathematics. His theoretical model of the hyper-mutation process served as inspiration for CLONALG [22], a popular AIS algorithm. In the 1980s, Framer et al. [35] specify the interaction between antibodies mathematically in which he formalized the idiotypic network model in combination with clonal selection. The network model provided network-based approach distinct from neural network and genetic algorithms. Another refinement to this model is implemented by Bersini and Varela [15]. These models form the basic of all AIS work. Timmis and Hart [163] argued that clonal selection algorithms are robust systems which make them popular, well-understood AIS algorithms and suited for complex optimization problems such as multi-objective optimization [55].

All clonal selection algorithms have a repeated cycle of match, clone, mutate and replace where large number of parameters can be tuned such as cloning rate, number of antibodies and the mutation rate [55].

In comparison to adaptive algorithms such as genetic algorithm, clonal selection algorithms have higher mutation rate. In addition, clonal selection algorithms are similar to K-nearest neighbour algorithm where K is equal to one and they combine the features of K-means where the position of centroids is adjusted. In contrast to these algorithms, clonal selection algorithm differs in terms of affinity metrics and hypermutation components [55].

Clonal selection algorithms have been used in many applications such as pattern recognition and optimization. For example, Opt-AI [29] is a hypermutation component of clonal selection algorithm where a dynamic local search is performed for the prediction of protein secondary structure. Another example is AIRS which is a multi-class classifier that contains a clonal selection component [55].

6.3.2 Negative Selection

Perelson et al. [17][137] presented a theoretical model of the selection of T-cell called a negative selection for the detection of computer viruses. Negative selection is one-signal supervised machine learning model which is used for anomaly detection. In this model, the detection of potential anomalous or non-self antigen is performed by the selection of T-cell to model the suitability of T-cell receptor (TCR) [55]. Later, Perelson in collaboration with Forrest [39] managed to transfer the theoretical immunology concepts to an AIS algorithm [48][47]. Pattern matching is used to discriminate between normal and abnormal data through the use of 'detectors'. Input data are represented as strings and a matching function is used to find similarities between strings. Negative selection has two phases, training phase and test phase. During the training phase, an initial population of detectors is created to represent randomly generated bit strings of a specified length. This training dataset consists of items which defines normal data. Each training item is compared with each detector. A detector is removed from the population if it matches sufficient numbers of normal items. The test data items are compared with the remaining detectors and an alert is generated when an incoming item matches a detector [55][47].

Negative selection has been used in many areas such as fault tolerance problems and computer security problems for the detection of executable computer viruses as an approach to discriminate between self-nonself [39]. When applied to computer security, the discrimination between self (i.e. normal) and nonself (i.e. abnormal) is based on building a profile of sequences of system calls during normal case. Any subsequent sequences that deviate from the existing sequences in the normal file is considered as a possible intrusion [55]. Such research was inspired by Forest et al. [38] and Hofmeyr and Forrest [67]. Kephart [89] tried to build such an approach by implementing a heterogeneous AIS based on self-nonself principle.

Another approach known as the true negative selection algorithm was introduced. This approach consists of three phases. The first phase is used to define a self for normal profile which represents normal behaviour patterns. The second phase generates a set of random detectors. The final phase compares each detector against all self patterns within the self profile. If there is no match, anomaly is detected [55].

Although optimistic results were obtained, it was a state-of-the-art algorithm and assumed to be out-dated, oversimplified in comparison to the actual functionalities of a human immune system. In 2004, a study by Cohen [25] showed that immune system is a complex adaptive system which leads to exchange of many ideas between theoretical and computational researchers and they can better cooperate in the field of AIS. Negative selection algorithms have a number of problems [147][91][90][48][47]. First, it suffers from the scalability issue [149] which affects it use within computer security. This appears when creating a large number of randomly-generated detectors which can be increased exponentially as the dimensionality of the feature space increases [91][150]. Second, the normal profile of negative selection algorithm is not updated over time, thus generating large number of false positive alarms due to incorrect classification of data. This is an important issue when it applies to the security area. The third issue is that negative selection algorithms can be applied to small and constrained problems, where the definition of normal profile does not change regularly over time [55]. The interested reader can refer to the review by Ji and Dasgupta [83]. As a result, the development of more sophisticated AIS algorithms becomes obvious.

6.4 Danger Theory Approaches

AIS researchers noticed that it was difficult to distinguish AIS approaches such as negative selection and clonal selection algorithms from machine learning techniques because of the similarities between the two approaches. Although these algorithms were used in many application areas such as pattern recognition, detection and classification, little further development was done on the algorithms. In addition, such algorithms had problems to deal with complex datasets in computer security. As a result, the need to re-design these algorithms into a complex and sophisticated model incorporating proper functionalities of immune system was important.

In order to overcome the problems of previous AIS algorithms such as negative selection algorithms which suffers from a scaling issue and produces large number of false positive, a danger project by Aickelin et al. [1] was started in 2003. The danger project is a collaboration between computer science researchers represented by the AIS team and immunologists. Their aim is to improve the first-generation of AIS by presenting complex properties desired of AIS algorithms and to apply them in different areas.

The aim of danger theory was to improve the performance of previous AISs by producing a robust intrusion detection system in the area of computer security that is capable of detecting anomalies in a real-time system with low false alarms. This project aimed to build abstract computational models of the cells involved in detection of danger signals to form a basic model of novel algorithms and frameworks. In their proposal, Aickelin et al. [1] suggest that secondary behavioural context signals could be incorporated into intrusion detection systems to reduce the number of false positive alarms. To apply the concept of danger project, two systems were developed [48][55][47].

The first development was the innate immune framework termed 'libtissue system' which is an agent-based framework [166][167][55]. Libtissue framework assumes that the cells of the immune system need to be in the correct place and certain types of cells should be kept separate from each other. This procedure allows libtissue framework to facilitate separate areas for processing within the immune algorithm. For example, the libtissue framework has two sections, the tissue which is responsible for the integration of signals and antigen and the lymph node for the processing of the interaction between T-cells and DCs [47].

Libtissue framework creates population of cells to interact with population of T-cells. Each cell is defined through a number of inputs (termed receptors), storage of internal values, the processing stage and the output stage from the cells (termed cytokine). Receptors define a cell as it can be processed while cytokines define a cell as it can be produced. The production of one type of cell by cytokines and the sensing of signals by receptors allow networking between individual cell populations.

In order to test this framework, different novel algorithms were developed. The first algorithm is the positive selection which is used to validate libtissue while the second algorithm is termed two-cell and it is used to test the facility to define multiple cell types and observe the interaction behaviour between these cells. Another algorithm termed toll-like receptor (TLR) is developed and consists of interaction population of T-cells and DCs [47]. In TLR, all signals are treated equally and are more related to PAMP signals to indicate selected signature of intrusion. The abstraction of signals is binary signals which represent the availability of signal value. TLR consists of two stages, training phase and testing phase. In training phase a list of discrete signals values (termed infectious signal list) are compiled. Once the DCs sense these signals, it activates the TLRs on the DCs. In the training phase, DCs transfer to mature states upon activation by any signal [47]. The system showed a success when applied to detect anomalous system calls captured from an exploited ftp server [166]. The results show that TLR performs well in comparison to Systrace [124] and additional signals to DCs can reduce the false positive alarms when compared to negative selection approach which lead to the creation of Dendritic Cell Algorithm (DCA) [46] as a part of the danger project.

6.4.1 Dendritic Cell Algorithm (DCA)

The DCA is a second generation algorithm based on the abstraction behaviour of dendritic cells and their ability to distinguish between normal and infected tissue first presented in [51]. In the human immune system, DCs are responsible for collecting and correlating molecular signals (apoptosis and necrosis) found on the tissue and use this information to assess the context of this area looking for the evidence of damage. In addition, DCs collect debris which is considered to be the suspect causing damage to the tissue. These suspects are termed 'antigen'. If DCs collect enough antigen after a period of time, they migrate from the tissue to the lymph node where they present their context information and antigen to a population of T-cell in which they react accordingly. More information about the functionality of natural DCs can be found on Lutz and Schuler review citelutz02.

The DCA is implemented using the libtissue framework to manage the updating of cells and the tissue attributes. DCA detects anomalies by performing anomaly detection though the process of correlation and classification. DCA has been applied to many areas such as intrusion detection [49], port scan detection [52][53], insider attack detection, standard machine learning intrusion datatsets, robotic security [117], schedule overrun detection in embedded systems and sensor networks. The results showed that the algorithm's performance is high and produces low number of false positives rates.

When viewed from a computational perspective, DCs are part of the innate immune system. DCs are considered as anomaly detector agents, which are responsible for data fusion and generating appropriate actions in response to the attack in the human body, acting as an interface between the innate and adaptive system.

In nature DCs exist in one of three states: *immature*, *semi-mature* and *mature* and the final state of these cells allows the adaptive system to decide whether or not to take action and respond to a potentially harmfull entity. The initial maturation state of a DC is immature for sensing and processing three categories of input signals (see Table 6.2) and in response produces three output signals. The three input signals can influence the behaviour of DCs sensitivity. The signal categorization is one important difference between the DCA and the TLR algorithm.

The first two input signals are S_1=*Pathogen Associated Molecular Patterns* (PAMP) and S_2=*Danger Signals* (DS) signals. S_1 signals are derived from the detection of pathogens while S_2 signals are generated from the unexpected cell death due to damage, to the tissue cells. The third input signal is S_3=*Safe Signals* (SS) which are molecules released as a result of normal cell death.

In terms of the algorithm, the DCA is a population based algorithm which performs anomaly detection based on the indication of abnormality of the system by performing filtering of input signals, asynchronous correlation of signals with suspect entities (termed antigen), and classification of antigen types as normal or anomalous. The term asynchronous correlation between signals and antigen means that the signals are updated in discrete manner while the antigen are updated in continues manner. The input signals are pre-normalized and pre-categorized data sources based on preliminary experimental data to reflect the behaviour of the system being monitored. Initially the cells are all assigned the ‘immature’ state label. While in this state, DCs capture the antigen and combine them with evidence of damage in the form of signals to provide information about how ‘dangerous’ a particular protein is to the host body.

The correlation between antigen and signals is performed at the individual cell level but the classification of antigen types is performed at the population level. Each cell’s input signals are transformed to cumulative output signals over time. Signal data enters the system and is stored in an array. Each time a cell is updated; these input signals are used to form a set of cumulative updated output signals. Each cell performs a weighted sum equation to produce interim output values. These interim values are added to final output values resulting in three output values per cell; the costimulatory signal (CSM), semi-mature cell (semi) and mature cell (mat). CSM is used to control and limit the DCs lifespan unlike the predefined values in TRL algorithm. Semi-mature cell and mature cell are used to determine the final state of the cell. If the DC, over its lifespan, has collected majority of S_3, it will change state to a semi-mature state and suppress the activation of the immune system. Conversely, cells exposed to S_1 and S_2 signals transforms into a mature state and can instruct the immune system to activate.

Signal processing occurs within DCs of the immature state. Each DC in the immature state performs three functions as follows:

- To sample antigen by collecting antigen from an external source and transfers the antigen to its own antigen storage facility.
- To update input signals in which the DC collects values of all input signals present in the signal storage area.
- To calculate temporary output signal values from the received input signals; Each DC generates three temporary output signal values from the received input signals per iteration. These temporary output values are added to obtain the cell's CSM, semi-mat and mat values.

The transformation from input to output signal per cell is performed using a simple weighted sum in order to reduce the computational overheads (Equation 6.1). The weighted sum is described in detail in [52] with the corresponding weights given in Table 6.1. These weights determine the value of the output and are derived from preliminary observation that defines the danger level of the input signals. The original values of signal weight is the WS_3 but in order to examine the effect of changing weight values on our detection performance, we introduce different weight sets. The values of weight sets (WS) ranges from $k = 1$ to $k = 5$ ($WS_k = WS_1$ to WS_5). As shown in Table 6.1, for the semi-mature cell, the weight values for PAMP, DS and SS remains unchanged for all sets. For CSM and mature-cell, we have increased and decreased the weight values for PAMP, DS and SS to the point where further increase or decrease

TABLE 6.1: Signal Weight Values.

	signal	WS_1	WS_2	WS_3	WS_4	WS_5
O_1=csm	S_1=PAMP	2	4	4	2	8
	S_2=DS	1	2	2	1	4
	S_3=SS	2	6	3	1.5	0.6
O_2=semi	S_1=PAMP	0	0	0	0	0
	S_2=DS	0	0	0	0	0
	S_3=SS	1	1	1	1	1
O_3 =mat	S_1=PAMP	2	8	8	8	16
	S_2=DS	1	4	4	4	8
	S_3=SS	-3	-12	-6	-6	-1.2

to these values does not have noticeable impact on the detection performance. The chosen

$$O_j = \sum_{i=1}^{3}(W_{ijk} * S_i) \qquad \forall j, k \tag{6.1}$$

where:

- W is the signal weight of the category i
- i is the input signal category ($S_1 = PAMP$, $S_2 = DS$ and $S_3 = SS$)
- k is the weight set index WS_k as shown in Table 6.1 ($k = 1 \ldots 5$)
- O_j is the output concentrations of one of the following signal:
  1. $j = 1$ costimulatory signal (CSM)
  2. $j = 2$ a semi-mature DC output signal (semi)
  3. $j = 3$ mature DC output signal (mat)

In the algorithm, the signal values are assigned real valued numbers and the antigen are assigned as categorical values of the object to be classified. The algorithm has three different stages, the initialization stage, the data processing and the analysis stage as shown in Figure 6.1.

In the initialization stage, the algorithm creates DCs population where each DC is assigned a random 'migration' threshold upon its creation to represent a limited time

TABLE 6.2: Signals Definition

Signal Name	Symbol	Definitions
Pathogen Associated Molecular Patterns	$S_1 : PAMP$	A strong evidence of abnormal/bad behaviour. An increase in this signal is associated with a high confidence of abnormality.
Danger Signal	$S_2 : DS$	A measure of an attribute which increases in value to indicate deviation from usual behaviour. Low values of this signal may not be anomalous, giving a high value confidence of indicating abnormality. The danger signal has less effect on the output signal than the PAMP signal.
Safe Signal	$S_3 : SS$	A measure which increases value in conjunction observed normal behaviour. This is a confident with indicator of normal, predictable or steady-state system behaviour. This signal is used to counteract the effects of PAMPs and danger signals and thus has negative impact on the output signals.
Inflammation	$S_4 : IS$	Acts as an amplifier value for the three signal categories PAMP, DS and SS.

window for data sampling which is considered as a lifespan of the cell. This process adds robustness and flexibility to the system and makes antigen detection possible in different time periods. The input data forms the sorted antigen and signals (S_1, S_2 and S_3) with respect to the time and passed to the processing stage. Antigen are fed into a storage area to be randomly selected at any period of time. Signals are fed into a signal matrix. Once it receives information, each DC performs an internal correlation between signals and antigen with respect to a specified time window determined by the migration threshold, signals and antigen. To cease data collection, a DC must have experienced signals, and in response to this express output signals. As the level of input signal experienced increases, the probability of the DC exceeding its lifespan

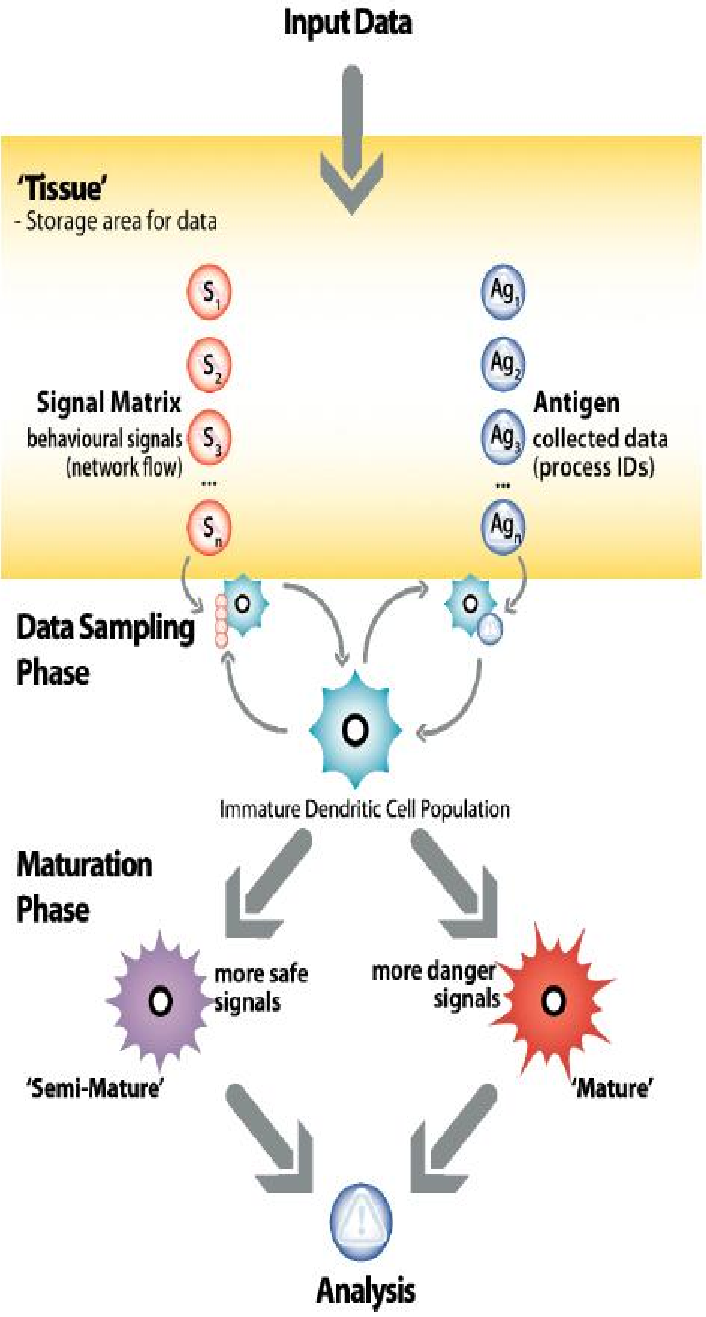

Figure 6.1: An overview of the DCA showing the input data (signals and antigen), the data sampling and maturation phases and finally the analysis stage which generates MCAV/MAC values. The above figure is taken from Greensmith Thesis [46].

also increases (i.e. The lifespan of each DC reduces whenever the CSM value increases since the CSM value is automatically derived from it). The level of input signal is mapped as a cumulative O_1 value. Once O_1 exceeds a migration threshold value, the cell ceases signal and antigen collection and presents all the collected antigen so that the semi-mat and mat values can be determined. After that, the cell is removed from the population and enters the maturation stage. Upon removal from the population the cell is reset (all internal values are set to zero) and immediately replaced by a new cell, to keep the population level static.

Because of the complexity of the natural mechanism of DC, a simple approximation of a thresholding mechanism using migration thresholds is used [48]. This approximation approach ensures that the DC has received sufficient information to present suitable context information in combination with antigen being collected during DC lifespan. As mentioned previously, each DC is assigned a random migration threshold. This range is based on Gaussian and uniform distribution to provide diversity for DCs population. A simple heuristic method is used to define the limits of threshold values range which relates to the median values of the input signal data. This allows different members of DC population to experience different sets of signals across a time window, thus, allows each DC to sample the signal matrix a different number of times throughout its lifespan. This process ensures that the same information is processed and assessed by different DCs. For example, if the input signals are kept constant, DCs with low migration threshold values sample for short period while DCs with high migration threshold values sample for a longer period producing a relaxed coupling between current signals and antigen.

A high concentration of S_1 and S_2 increases the probability of immature cells to become mature cells while a more concentration of S_3 impose the immature cells to become semi-mature cells. Therefore, if $O_2 > O_3$, the DC is termed a ‘semi-mature’ cell. Antigen presented by a semi-mature cell is assigned a context value of zero which represents ’safe’ context. In contrast, $O_2 < O_3$ leads to a ‘mature’ cell and antigen presented by a mature cell is assigned a context value of one which represents ’dangerous’ context. The detection of anomaly is based on having more mature cells than semi-mature cells in which the antigen in a mature context is detected. The algorithm runs until no further data is available where it terminates its processing. The pseudo code for the functioning of cells is presented in Algorithm 3.

In previous experiments with the DCA, the system calls invoked by running processes are used as antigen [49]. This implies that behavioural changes observed within the signals are potentially caused by the invocation of running programs. For the purpose of bot detection, antigen are derived from API function calls, which are similar to system calls. The resultant data is a stream of potential antigen suspects, which are correlated with signals through the processing mechanisms of the DC population. One constraint on antigen is that more than one of any *antigen type* must be used to be able to perform the anomaly analysis with the DCA. This will allow for the detection of which type of function call is responsible for the changes in the observed input signals.

Analysis

Each time input signals are received an antigen may also be collected. All antigen collected by a cell during its lifespan must be presented in conjunction with context value. Antigen are collected from the antigen vector and stored until presentation where any modification of antigen by cells is prohibited. A minimum of ten cells are required to perform processing [50]. Once all antigen and signals are processed by the cell population, an analysis stage is performed. This stage involved calculating an anomaly coefficient per antigen type - termed the *mature context antigen value*, MCAV. The derivation of the MCAV per antigen type in the range of *zero* to *one* is shown in Equation 6.2. The closer this value is to one, the more likely the antigen type is to be anomalous. The creation of MCAV adds robustness to the system because it cancels out any errors made by individuals in the DC population.

$$MCAV_x = \frac{Z_x}{Y_x} \tag{6.2}$$

where $MCAV_x$ is the MCAV coefficient for antigen type x, Z_x is the number of mature context antigen presentations for antigen type x and Y_x is the total number of antigen presented for antigen type x.

```
input  : Sorted antigen and signals ($S_1$ :PAMP,$S_2$ :DS,$S_3$ :SS)
output: Antigen and their context (0/1)

Create DC population size of 100;
Initilize DCs;
foreach cell in DC population do
    randomly select 10 DCs from population;
    for Selected DCs 1 to 10 do
        get antigen;
        store antigen;
        get signals;
        calculate interim output signals;
        update cumulative output signals;
        if CSM output signal ($O_1$) $>$ migration threshold then
            DC removed from population;
            DCs context is assigned;
            if semi-mature output ($O_2$) $>$ mature output($O_3$) then
                cell context is assigned as 0;
            else
                cell context is assigned as 1;
            end
            All DCs collected antigen and context is output for analysis;
            new DC is added to population;
        else
            DC is returned to population for further sampling;
        end
    end
    generate MCAV per antigen type;
end
```

Algorithm 3: DCA algorithm

In the previous work [46], it has been shown that the MCAV for processes with low numbers of antigen per antigen type can be higher than desired. This can lead to the generation of false positives. In this work we address this problem by introducing an anomaly coefficient which is an improvement on the MCAV, by incorporating the number of antigen used to calculate the MCAV. This improvement is termed the *MCAV Antigen Coefficient* or MAC. The MAC value is the MCAV of each antigen type multiplied by the number of output antigen per process and divided by the total number of output antigen for all processes. This calculation is shown in Equation 6.3. As with the MCAV, the MAC value also ranges between *zero* and *one*. The closer the MAC value to *one*, the more anomalous the process is.

$$MAC_x = \frac{MCAV_x * Antigen_x}{\sum_{i=1}^{n} Antigen_i} \tag{6.3}$$

where $MCAV_x$ is the MCAV value for process x and $Antigen_x$ is the number of antigen processed by process x.

Pros and Cons of DCA

One of the main advantages of using DCA in computer security area is that it has very low CPU processing requirements. In addition, no training phase is required. Since DCA is derived from an abstract model of natural DC function, it provides robust detection and correlation. Unlike previous AIS, the DCA does not perform pattern matching on the actual value of the antigen but the classification is based on signals processing at the time of antigen collection [47]. In contrast to negative selection algorithm, the DCA does not have an adaptive component, which removes a formal training phase from the system. DCA is based on large number of probabilistic components such as random sorting of the DC population, random selection of cells, variable threshold, the probability of antigen collection and decay rates of signals which make the system very difficult to analyze. In addition, a proper selection of antigen is required to perform a pre-classification of input signals which makes it differ from a negative selection algorithm. This is because the DCA relies on heuristic-based signals that are not absolute representation of normal or anomalous. Although the

DCA appears to be as a neural networks algorithm, the variable lifespan, dynamic population and filtering, correlation and classification functionality make it differ from other approaches.

DCA Parameters

DCA consists of many tunable parameters such as number of cells, maturation threshold, the number of input signals, weights values for processing input signals, and other parameters. The values of some of these parameters are defined in Table:

The parameters that we have used in the algorithm are as follows:

- Number of categories of signal (J) = 4 (PAMP, DS, SS, Inflammation);
- Number of signal per category (I) = 1;
- Maximum number of antigen in tissue antigen vector (K) = 500;
- Number of DC cycles (L) = 120;
- Population size (M) = 100;
- DC antigen vector size (N) = 50;
- Number of output signals per DC (P) = 3;
- Number of antigen sampled per DC per cycle (Q) = 1;
- Number of Antigen receptors (R) = 10;

For in depth details of the algorithms and parameters used, the interested reader can refer to Greensmith thesis [46].

6.5 Methodology

6.5.1 Introduction

In the previous chapter, we saw that the Spearman's rank correlation algorithm suffers from many limitations such as the existence of large number of idle periods which affects the detection performance by generating many false positive alarms. In order to reduce these false alarms, a more intelligent way of correlating data is needed. One of these algorithms introduced by Greensmith [46] is called the Dendritic Cell Algorithm (DCA).

The DCA is an intelligent way of data fusing and correlating information from disparate sources. The information from disparate sources is correlated with potentially anomalous 'suspect entities'. As a result, we have collected our data according to the DCA input format. We have applied the collected data to both the Spearman's rank correlation algorithm and the DCA. The results from the Spearman's rank correlation, will only show if an anomaly is detected in the information. On the other hand, the DCA will not only state if anomaly is detected in the information but it will provide the culprit responsible for the anomaly.

The aim of this work is to investigate the effect of correlating bot's behavioural attributes of the processes and generates the status of these processes by applying new correlation algorithm (i.e DCA) in comparison to the Spearman's rank correlation algorithm to the detection of a single bot. Different experiments have been conducted to achieve this task. For these experiments the basis of classification is facilitated through the correlation of different activities such as keystrokes interception, how fast the program executes certain communication function calls and how fast does the program react when receiving information.

For the purpose of experimentation two different types of bot are used, namely spybot [12] and sdbot[162]. According to Overton [118], these two bots constitute a high percentage of malicious bots. The spybot is a suitable candidate bot as it uses a range of malicious functionalities such as keylogging and SYN attacks which are frequently used features by bots. The sdbot is also used as it contains the additional functionality of a UDP attack. An IRC client, IceChat [77], is used for normal conversation and to send files to a remote host which represents 'normal' traffic. To provide suitable data for the DCA an APITrace intercepting program is implemented to capture the required behavioural attributes by intercepting specified function calls. The collected data is processed by both the Spearman's rank correlation algorithm and the DCA to measure the detection performance.

We also investigate various sensitivity analyses of the signal weights. In addition, we also introduce the MAC value as a replacement to the existence MCAV.

The results show that the DCA is a better correlation algorithm as compared to the Spearman's rank correlation. In addition, we have noticed that using the MAC value generates better results compared to the MCAV value introduced by Greensmith for this type of experiments.

6.5.2 Bot Scenarios

To emulate real-world bot infections, three different scenarios are constructed including inactive (E1), attack (E2.1-2.3) and normal (E3) scenarios. The attack scenario consists of three sessions: a keylogging attack session, a flooding session and a combination session comprising both keylogging and packet flooding.

- *Inactive bot (E1):* This session involves having *inactive bots* running on the monitored host in addition to normal applications such as an IRC client, Wordpad, Notepad and terminal emulator (CMD) processes. Spybot is used for this session. The bot runs on the monitored victim's host and connects to an IRC server and joins a specified channel to await commands from its controller, though no attacking actions are performed by this idle bot. This results in minimal data, with the majority of transactions involving simple PING messages between the bot, the IRC server and the IceChat IRC client.

- *Keylogging Attack (E2.1):* The spybot is capable of intercepting keystrokes using various methods, upon receipt of the relevant command from the botmaster. In this scenario, two methods of keylogging are used including the "GetKeyboardState" (E2.1.a) and "GetAsyncKeyState" (E2.1.b) function calls. However, detection cannot be performed by examining these two function calls alone, as some of the legitimate programs often rely on such function calls. For example, MS Notepad utilises GetKeyboardState as part of its normal functioning. The DCA will be employed to discriminate between malicious and legitimate keystroke function calls.

- *Flooding Attack (E2.2):* This involves performing packet flooding using the spybot for a SYN flood attack (E2.2.a) and the sdbot for a UDP attack (E2.2.b). These flooding methods are designed to emulate the behaviour of a machine partaking in a distributed denial of service attack. As part of the process of packet flooding the bots rely heavily on socket usage, as part of the packet sending mechanism. Therefore to detect these attacks, socket usage monitors are employed, with the exact nature of this data given in the forthcoming section. It is important to note that during the flooding attack no 'normal' legitimate applications are running.

- *Combined Attack (E2.3):* In this session, both keylogging and SYN flooding (SYN flood [E2.3.a] and UDP flood [E2.3.b]) are invoked by the bot. As with session E1, spybot is used to perform this attack. Note that the two activities can occur simultaneously in this scenario.

- *Normal Scenario (E3):* The normal scenario involves having normal conversation between the two parties. It also includes transferring a file of 10 KB from one host to another through IRC client. Other applications such as Wordpad, Notepad, cmd and the APITrace intercepting program are running on the victim's host. Note that no bots are used in this scenario.

6.5.3 Signals

Signals are mapped as a reflection of the state of the victim's host. Three signal categories are used to define the state of the system namely S_1:PAMP, S_2:danger signal (DS) and S_3:safe signal (SS), with one data source mapped per signal category. The mapping of raw signals to signals for DCA is determined via expert knowledge. These signals are collected using a function call interception program. Raw data from the monitored host is transformed into log files, following a signal normalisation process. The normalisation process for each signal is based on a pre-defined maximum value for each signal. The resultant normalised signals are in the range of 0 - 100 for S_1 and S_2 with S_3 having a reduced range, as suggested by Greensmith et al. [52]. This reduction in S_3 ensures that the mean values of each signal category are approximately equal, with preliminary experiments performed to verify this.

In terms of the signal category semantics, S_1*:PAMP signal* is a strong evidence for bad behaviour on a system. Because we focus on detecting bots performing keystrokes interception in combination with other malicious activities, we have used this activity as our S_1. This signal is derived from the *rate of change* of invocation of selected API function calls used for keylogging activity. Such function calls include GetAsyncKeyState, GetKeyboardState, GetKeyNameText and keybd_event when invoked by the running processes. To use this data stream as signal input, the rate values are normalised. For this process n_{ps}, (ps is referred to the PAMP signal), is defined as the maximum number of function calls generated by pressing a key within one second. Through preliminary experimentation it is shown that by pressing any key on the keyboard for a duration of one second, a n_{ps} number of calls are generated, in our case $n_{ps} = 25$ and it is mapped to 100 as a maximum value of S_1. Subsequently n_{ps} is set to be the maximum number of calls that can be generated per second. The normalized S_1 signal applies a linear scale between 0 and 100.

An example of S_1*:PAMP signal* normalization process is shown below:

$$\begin{aligned}
PAMP_0 &= 0 \\
PAMP_1 &= 2 \\
PAMP_2 &= 10 \\
PAMP_3 &= 20 \\
PAMP_4 &= 20 \\
... &= ... \\
PAMP_t &= P_{amp}
\end{aligned}$$

where P_{amp} is the number of keyboard status function calls generated during that period and t is the time. Therefore, we calculate the change of S_1*:PAMP signal* from the following:

$$PAMP_t = \frac{PAMP_t - PAMP_{t-1}}{n_{ps}} * 100 \ldots \forall t$$

In our example:

$$\begin{aligned} PAMP_1 &= \tfrac{(2-0)}{(25)} * 100 = 8.0 \\ PAMP_2 &= \tfrac{(10-2)}{(25)} * 100 = 32.0 \\ PAMP_3 &= \tfrac{(20-10)}{(25)} * 100 = 40.0 \\ PAMP_4 &= \tfrac{(20-20)}{(25)} * 100 = 0.0 \end{aligned}$$

The *danger signal*(S_2) is derived from the time difference between receiving and sending data through the network for each process by intercepting the send() and recv() function calls. As bots respond directly to botmaster commands, a small time difference between sending and receiving data is observed. In contrast, normal chat will have a higher value of time difference between sending and receiving activity. As with S_1 signal, the normalisation of S_2 involves calculating a maximum value. For this purpose n_{ds}, (*ds* is referred to the danger signal), is the maximum time difference between sending a request and receiving a feedback. If the time difference exceeds n_{ds}, the response time is normal. Otherwise, the response time falls within the abnormality range.

We set up a critical range (0 to n_{ds}) that represents an abnormal response time. The zero value is mapped to 100 *max-danger* time and n_{ds} is mapped to zero *min-danger* time. If the response time falls within the critical value (in our case, the critical value is from 0 to 50 seconds), it means that the response is fast and considered to be dangerous.

For example: if the time difference between *recv* and *send* is less than or equal to 50 seconds, it is calculated using the following formula:

$$D_n = 100 * (1 - (T_{recv,send}/n_{ds}))$$

Where D_n is the normalised *danger signal*(S_2) value, $T_{recv,send}$ is the time difference between executing recv and send function calls and n_{ds} is the critical danger signal and it is equal to 50. Otherwise the value of danger signal D_n is set to *zero.*

Finally, S_3*:safe signal* is derived from the time difference between two outgoing consecutive communication functions such as [(send,send),(sendto,sendto),(socket,soc ket)]. This is needed as the bot sends information to the botmaster using *send* function call or issues SYN or UDP attacks using *sendto* or *socket* which generates many function calls within a short time period. Therefore we set n_{ss1} and n_{ss2} (ss is referred to SS signal) as a range of a time difference between calling two consecutive communication functions. If the time difference is less than n_{ss1}, the time is classified within a *min-safe* time. If the time difference falls between n_{ss1} and n_{ss2}, the time is classified as uncertain time. If the time difference is more than n_{ss2}, the time is classified as *max-safe* time. These timings are scaled between 0 and 10. By recording the time that a bot takes to respond to the command in most of the experiments that we have conducted, we have noticed that the mean value for bot to respond to the command is around 3.226 seconds. Therefore, we set up a critical range for S_3 signal. We divide our critical range into three sub-ranges. The first range is from *zero* to n_{ss1} where $n_{ss1} = 5$ to allow enough time for a bot to respond to the attack's command. Any value that falls within this range is considered as an *min-safe* time. The second range is where there is uncertainty of response. The uncertainty range is between n_{ss1} and $n_{ss2} = 20$. The third range is that the time difference is above n_{ss2} and is considered as a *max-safe* time. In this range, we are sure that the time difference between two consecutive function calls is generated as a normal response.

The *safe signal*(S_3) is mapped as following:

$$SS = \begin{cases} \Delta T * 0.2 & \textit{if } \Delta T \in [0,5] \\ \Delta T * 0.5 & \textit{if } \Delta T \in]5,20] \\ \Delta T & \textit{if } \Delta T \in]20,\infty] \end{cases} \tag{6.4}$$

In case of S_2*:danger* and S_3*:safe* signals, this decision is based on the assumption that the attacker designs the bot to respond to his/her commands without adding a short random delay when responding to the commands or when flooding other hosts or network.

6.5.4 Antigen

For the purpose of bot detection, antigen are derived from API function calls, which are similar to system calls in Unix/Linux environment. The resultant data is a stream of potential antigen suspects, which are correlated with signals through the processing mechanisms of the DC population. One constraint on antigen is that more than one of any *antigen type* must be used to be able to perform the anomaly analysis with the DCA. This will make it possible to detect which type of function call is responsible for the changes in the observed input signals.

The collected signals are a reflection of the status of the monitored system. Therefore, antigen are potential culprits responsible for any observed changes in the status of the system. The correlation of antigen signals is required to define which processes are active when the signal values are modified. Any process executed one of the selected API function calls explained in section 3.6, the process identification (ID) which causes the execution of the function call is stored as an antigen in the antigen log file. The more active the process, the more antigen it generates. Each intercepted function call is stored and is assigned the value of the process ID to which the function call belongs and the time at which it is invoked.

For the Spearman's rank correlation experiments, only the signals (S_1, S_2, S_3) log file is used to detect the malicious activities. In case of DCA, signal and antigen logs are combined and sorted based on time. The combined file forms a dataset which is passed to the DCA through a data processing client. The combined log files are parsed and the logged information is sent to the DCA for processing and analysis.

6.5.5 Data Collection

It is assumed that the bot is already installed on the victim's host, through an accidental 'trojan horse' style infection mechanism. Therefore, we are not attempting to prevent the initial bot infection but to limit its activities whilst on a host machine (i.e. uses an extrusion detection system). The bot runs as a process whenever the user reboots the system and attempts to connect to the IRC server through IRC standard ports (in the range of 6667-7000). The bot then joins the IRC channel and waits for the botmaster to login and issue commands.

An interception program is implemented and run on the victim's machine to collect the required data. Two types of log files are produced, SigLog and AntigLog. The SigLog presents values of S_1 :PAMP, S_2 :danger signal (DS), S_3 :safe signal (SS) and S_4 : Inflammatory signal which always has a zero value in the following format as mentioned in the example below. The inflammatory signal is used to amplify the other three signals but because we assign a zero value for this signal, there is no effect of this signal on the other signals.

```
<time> <type> <fixed> <# of signals> <S1:PAMP> <S2:DS> <S3:SS> <S4:Inf>
<0001> <signal> <0> <4> <3> <11> <32> <89> <0>
```

The AntigLog presents the intercepted API function calls with respect to its process ID (PID) in the following format:

```
<time> <type> <PID> <# of antigen> <Function call name>
<0002> <antigen> <722> <1> <GetAsyncKeyStat()>
```

After finishing the data collection, the SigLog is passed to Spearman's rank correlation algorithm for analysis. In case of DCA, the SigLog and AntigLog are merged together and sorted with respect to the time and the combined file is passed to the DCA for the analysis.

Three specific types of function calls are used as signal and antigen input to the DCA. These function calls are Communication functions, File access functions and Keyboard status functions which are discussed in section 3.6.

The communication functions are used because the bots needs to communicate with the botmaster in order to send or receive information. In addition, these function calls are used in flooding attack. The file access functions are needed because once a bot intercepts the user keystrokes, it needs to store the intercepted data in a buffer or in a file for future access. The keyboard status functions are needed because many existing bots implement the keystrokes logging by executing these functions in 'user mode' level in windows environment. Invoking these function calls within specified time-window can represents a security threat to the system, but may also form part of legitimate usage.

6.6 Experiments

The aim of these experiments is to compare the results of Spearman's rank correlation and the DCA when applying the collected datasets and to measure their performance in detecting the bot running on the system. Various experiments are performed to verify this aim. Each experiment is repeated ten times which is sufficient, as the results from the repeated experiments produce a small variation on standard deviation by using Chebyshev's Inequality. After collecting and processing the data, one dataset is selected randomly from each repeated experiments. The dataset is passed to both the Spearman's rank correlation algorithm and the DCA.

1. *Null Hypothesis One (H1):* The data collected in each dataset is normally distributed. The Shaprio-Wilk test is used for this assessment.

2. *Null Hypothesis Two (H2):* The Spearman's rank correlation algorithm is able to detect the existence of bot when correlating different attributes.

3. *Null Hypothesis Three (H3):* The DCA algorithm using the MCAV/MAC values for the normal processes are not statistically different from those produced by the bot process. This is verified through the performance of a two-sided Mann-Whitney test.

4. *Null Hypothesis Four (H4):* Variation of the signal weights in DCA algorithm as described in Table 6.1 produces no observable difference in the resultant MCAV/MAC values and the detection accuracy. Wilcoxon signed rank tests (two-sided) are used to verify this hypothesis.

5. *Null Hypothesis Five (H5):* There is no difference between the Spearman's rank correlation algorithm and DCA in terms of performance on detecting bot.

In all DCA experiments, the parameters used are identical to those implemented in [52], with the exception of the weights. The statistical analyses are performed using R statistical computing package (v.2.6.0).

6.7 Results and Analysis

Upon the application of the Shapiro-Wilk test to each of the datasets, the resultant p-values imply that the distribution of the datasets is not normal. Therefore, the null hypothesis one (H1) is rejected. As a result of this, further tests with these data use non-parametric statistical tests such as the Mann-Whitney test, also using 95% confidence.

6.7.1 Spearman's Rank Correlation

We have used the Spearman's rank correlation (SRC) algorithm (described in Algorithm 2 in Chapter 5) to detect the bot with modified signals in order to evaluate the performance of SRC and DCA when using the same input data. These signals represent the input to the SRC and are described below:

- S_1: keystrokes interception.
- S_2: how fast the bot responds to attacker commands.
- S_3: how fast the bot repeats the same communication function calls.

In order to detect a bot in a system, different bot behaviours are correlated to generate a high correlation value represented by SRC value. Such behaviours include intercepting user keystrokes, how fast the bot responds to the commands of the attacker and how fast it executes same function calls. In our case, if SRC value exceeds a certain threshold level, a high correlation between the two different behaviours is generated. According to SRC algorithm, the threshold level of ±0.5 or higher represents a strong correlation between two events. This is the same threshold that is used in section 5.3.2.

The aim of SRC experiments is to verify the notion that correlating different behaviours of a single process indicates abnormal activity. In addition, using the same data ($PAMP$:S_1, DS:S_2 and SS:S_3) for both Spearman's rank correlation and the DCA allows for a better comparison between the two algorithms. We apply the monitoring and correlation scheme to a normal application to verify that the normal application behaves differently from the malicious process which results in having different correlation value.

Our hypothesis is that calling GetAsyncKeyState() or GetKeyboardState() functions by an unknown running program may represent abnormal behaviour in our system. This is because many of the current logging techniques in *user mode* level in windows environment use these two function calls to perform keylogging activities. However, we consider that calling these functions generates only a 'weak' alert because other legitimate programs may use the same API function calls. Therefore, the correlation of different types of bot behaviour is needed to enhance the detection confidence to generate a 'strong' alert.

In our experiments, we use SRC algorithm to correlate two different datasets. The data has been collected for the duration of one hour. Each data is generated every second leading to 3600 as the total number of raws in the dataset. The first dataset is PAMP and SS signals (S_1, S_3) dataset while the second dataset is DS and SS signals (S_2, S_3) dataset. In both datasets, we compare S_1 and S_2 with S_3 because the existing of S_3 suppress the effect of other two signals.

We analyse the results of the experiments described in Section 6.5.2. Table 6.3 represents the SRC value between the two datasets, (S_1, S_3) and (S_2, S_3), in each experiment. In this table, we have two sets of results. In set *Set1*, we correlate all the captured data from our algorithm including the idle period. In this period, no activity is noticed so therefore we assign a zero value to this period. This is represented by the *with zero* columns. In set *Set2*, we remove all the idle periods which have zeros and apply the SRC algorithm to the new data. The reason for having the two sets is that the idle periods in our data increase the correlation value. This is because there are many places where no activity is noticed in both datasets, which may produce inaccurate correlation. Therefore, we wanted to investigate the effect of having no idle periods.

The *Keylogging Activity* column represents the situation where the process calls any function used to intercept the keystrokes. As a result, we classify our detection scheme into three cases:

- *Normal detection (Normal)*: Keylogging activity is not detected and either low or high correlation value is noticed.
- *Medium detection (Medium)*: Keylogging activity is detected and a high correlation is noticed in one dataset.

- *Strong detection (Strong)*: Keylogging activity is detected and a high correlation is noticed in both datasets.

As mentioned in Section 5.3.2, a high correlation is considered if the SRC value exceeds the threshold (±0.5). From Table 6.3, we see a high correlation value between (S_1, S_3) and (S_2, S_3) in experiment *E1.* This is because the bot was inactive during all the time period. The only traffic generated by the bot is the PONG message to avoid disconnection from the IRC server. Therefore, the correlation value is expected to be high as well. We consider this situation as a 'normal' case.

In experiment *E2.1.a/b*, the bot intercepts the user keystrokes and sends the data to the botmaster. As a result, a high correlation value is expected and 'strong' detection is generated.

In experiment *E2.2.a/b*, we notice a high correlation value on both datasets. This situation is expected because the attacker issues a SYN attack and a UDP attack to participate on DDoS. The bot responds by generating a large number of same communication function calls for a long period. No keylogging activity is detected during this period. As a result, a 'normal' case is indicated. This situation represents the false negative case as it is incorrectly classified as normal.

Experiment *E2.3.a/b* shows a combined keylogging activity and SYN/UDP attack activity to participate on the DDoS. The correlation value of (S_1, S_3) is low compared to experiment E2.2.a/b. This is because the bot is intercepting keystrokes and performing the SYN/UDP attack simultaneously. As a result, the two datasets were noisy which generates a 'medium' detection case.

The last experiment *E3* shows the result of applying SRC algorithm on the IceChat client. Even though we have a high correlation value before and after removing idle periods on both experiments, we did not detect the use of keylogging function calls.

In summary, we notice that the SRC algorithm can detect activities which are happing simultaneously. For example, keylogging activities and a bot participating on DDoS by issuing a SYN attack or UDP attack. Although we have obtained optimistic results, some experiments produce low correlation values. There are many reasons for this. The first reason is that different events occur in different time-windows. As a result, SRC algorithm produces inaccurate results. The second reason is that some signals are varying differently influencing the correlation value and another reason is that we have many idle periods in our datasets, increasing the correlation value which affects our detection scheme. To improve this, we need to apply a more intelligent correlation scheme, as described in the next section. As a result, we cannot reject or accept the Null Hypothesis Two (H2) as we need a strong correlation algorithm to perform a better indication of malicious behaviour.

TABLE 6.3: Spearman's Rank Correlation (SRC) value which represents the correlation between two datasets.

$Exper-iments$	$SRC(S1, S3)$		$SRC(S2, S3)$		$Keylog.$ $Activity$ $existence$	API $Detection$ $confidence$
	$with$ $zeros$ $(Set1)$	$without$ $zeros$ $(Set2)$	$with$ $zeros$ $(Set1)$	$without$ $zeros$ $(Set2)$		
$E1$	0.987	0.727	0.966	0.878	No	$Normal$
$E2.1.a$	0.613	0.856	0.749	0.693	Yes	$Strong$
$E2.1.b$	0.621	0.879	0.754	0.745	Yes	$Strong$
$E2.2.a$	0.642	0.519	0.608	0.597	No	$Normal$
$E2.2.b$	0.554	0.504	0.538	0.512	No	$Normal$
$E2.3.a$	0.115	0.178	0.507	0.528	Yes	$Medium$
$E2.3.b$	0.205	0.326	0.587	0.572	Yes	$Medium$
$E3$	0.995	0.500	0.976	0.588	No	$Normal$

6.7.2 DCA

After using the Spearman's rank correlation algorithm, we tried a more intelligent way of correlating our signals using the dendritic cell algoritm - DCA.

The results from the DCA experiments are shown in Tables 6.4, 6.5 and 6.6. The mean MCAV and the mean MAC values for each process are presented, derived across the ten runs performed per scenario.

We start by checking the normality of our datatsets. Upon the application of the Shapiro-Wilk test to each of the datasets, it was discovered that the resultant p-values

TABLE 6.4: The results of the MCAV/MAC values generated from DCA using signal weight WS_3. The values that have asterisks are not significant

Experiment	Process	Processed Antigen	mean		Mann-Whitney P-Value	
			MCAV	MAC	MCAV	MAC
E1	bot	35	0.0978	0.0578		
	IRC	24	0.0625	0.0255	0.1602*	0.0202
E2.1.a	bot	1329.7	0.4736	0.4542		
	IRC	59	0.2881	0.0122	0.0002	0.0002
E2.1.b	bot	1296.2	0.5441	0.3098		
	IRC	464.9	0.5284	0.1077	0.0089	0.0002
	cmd	8.9	0.7889	0.0031	0.0002	0.0002
	Notepad	239.4	0.6916	0.0726	0.0002	0.0002
	Wordpad	268.8	0.8286	0.0977	0.0002	0.0002
E2.2.a	bot	19206.3	0.6047	0.6038		
	IRC	18	0.3441	0.0003	0.0002	0.0000
	cmd	9.8	0.2889	0.0002	0.0003	0.0000
E2.2.b	bot	5790.5	0.4360	0.4346		
	IRC	19	0.2772	0.0009	0.0002	0.0000
E2.3.a	bot	41456	0.8218	0.8214		
	IRC	20.5	0.5480	0.0003	0.0002	0.0000
E2.3.b	bot	22446	0.9598	0.9461		
	IRC	59.1	0.7802	0.0021	0.0000	0.0002
	cmd	9.7	0.6300	0.0003	0.0002	0.0002
	Notepad	23.1	1.0000	0.0010	0.0001	0.0002
	Wordpad	233.6	0.8801	0.0090	0.0002	0.0002
E3	IRC	135.5	0.1136	0.1136	N/A	N/A

imply that the distribution of scenarios E1, E2.1 and E3 are normal, with E2.2 and E2.3 being not normal. Given these two rejections of the null, Null Hypothesis One (H1) is rejected. As a result of this, further tests with these data use non-parametric statistical tests such as the Mann-Whitney test, also using 95% confidence.

For all scenarios E1-E3, a comparison is performed using the results generated for the bot versus all other normal processes within a particular session as shown in Table 6.4. In this table, the processed antigen column represents the number of antigen being processed by DCA. Because we have repeated each experiment ten times, we have considered the mean value for the total number of antigen being processed for all experiments. This is shown by the decimal points in the processed antigen column. The computed p-values using an unpaired Mann-Whitney test are presented, with those results deemed not statistically significant marked with an asterisks. In experiment E1, significant differences do not exist between the resultant MCAV values for the inactive bot and the normal IRC process, and so for this particular scenario the Null Hypothesis Three (H3) cannot be rejected. This can be attributed to the fact that the total number of antigen produced by both processes is too small in number to give an accurate description of the state of the monitored host. This is supported by the fact that the MAC values differ significantly for this experiment. This implies that the MAC is a useful addition to the analysis as it allowed for the incorporation of the antigen data, which can influence the interpretation of the results.

Significant differences are shown by the low p-values presented in Table 6.4 for experiments E2.1.a and E2.1.b for both the MAC and MCAV coefficient values, where the sample size is equal to ten. The differences are further pronounced in the generation of the MAC values, further supporting its future use with the DCA. We can conclude therefore, that the DCA can be used in the discrimination between normal and bot-directed processes and that the DCA is successful in detecting keylogging activities. This trend is also evident for scenarios E2.2.a/b and E2.3.a/b, where the bot process MCAV and MAC are consistently higher than those of the normal processes including IRC and notepad. This information is also displayed in Figure 6.2, Figure 6.3, Figure 6.4 and Figure 6.5 respectively. This implies that in addition to the detection of the bot itself the DCA can detect the performance of outbound scanning activity. Therefore the Null Hypothesis Three (H3) can be rejected as in the majority of cases the DCA successfully discriminates between normal and bot processes, with the exception of E1 because of the extrusion approach that we are taking.

Table 6.5 and Table 6.6 include the results of the sensitivity analysis on the weight values for the bot process. The aim of these experiments is to examine the effect of varying weight signals on to the DCA detection performance. Different values have been generated randomly to see the effect of increasing or decreasing S_1,S_2 and S_3 weight signals for O_1 =csm, O_2=semi-mature and O_3=mature cell. For example, in case of O_1, we have increased and decreased the weight value of S_1, S_2 and S_3 to

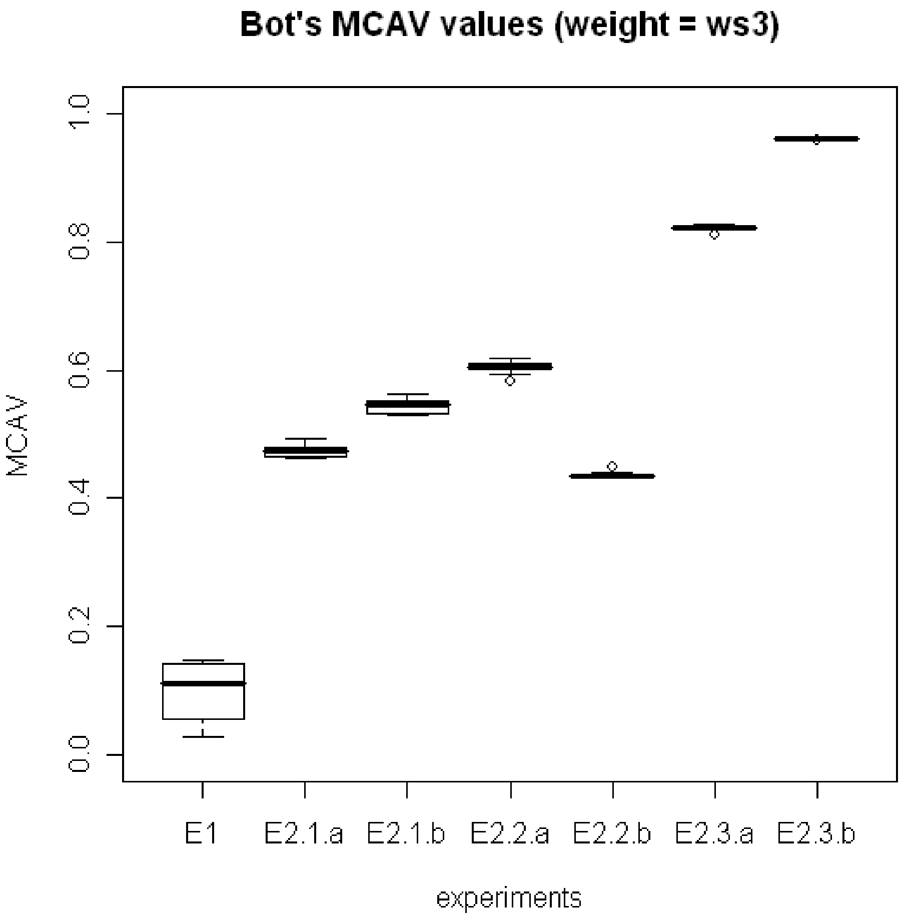

FIGURE 6.2: Bot's MCAV values generated by DCA using signal weight WS_3.

the point that reaches the steady state where further increase and decrease to these values will not have a large impact on the MCAV/MAC values.

The values presented in Table 6.5 and Table 6.6 are mean values taken across the ten runs per session (E1-E2). An arbitrary threshold is applied at 0.5: values above this threshold deem the process anomalous, and below as normal. From this data, it is shown that changing the weights used in the signal processing equation has significant effect on the performance of the system. For example, in the case of session E2.1.a, weight set WS_1 produces a MAC value of 0.09 for the bot yet produces a value of 0.72 for WS_5. This increase is likely to reduce the rate of false negatives.

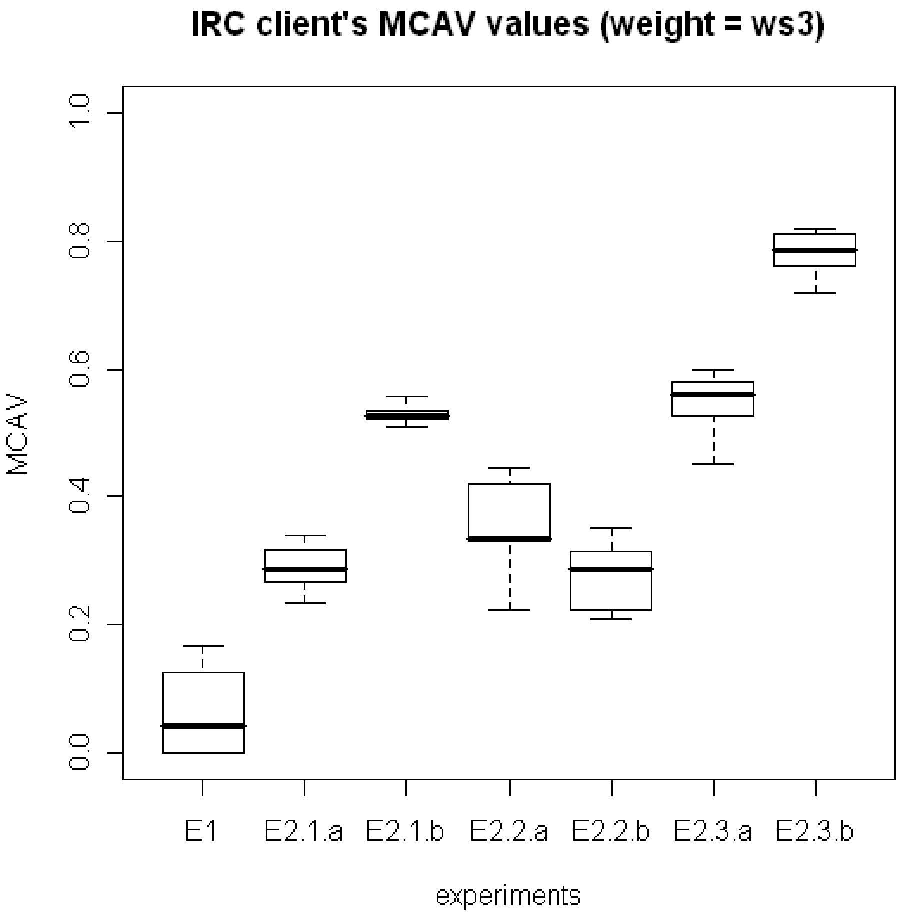

FIGURE 6.3: IRC client's MCAV values generated by DCA using signal weight WS_3.

To further explore these effects, the resultant data is plotted as boxplots as the data is not normally distributed. To assess the performance of the DCA as an anomaly detector the results for the anomalous bot and the normal IRC client are shown for the purpose of comparison. For these boxplots, the central line represents the median value, with the drawn boxes representing the interquartile ranges.

In Figure 6.6 and Figure 6.7 the median MCAV values are presented, derived per session across the ten runs performed for each WS (n=50). For the bot process in Figure 6.6, the MCAV is low for session E1, in-line with previous results. For E1, variation in the weights does not influence the detection results, as this process

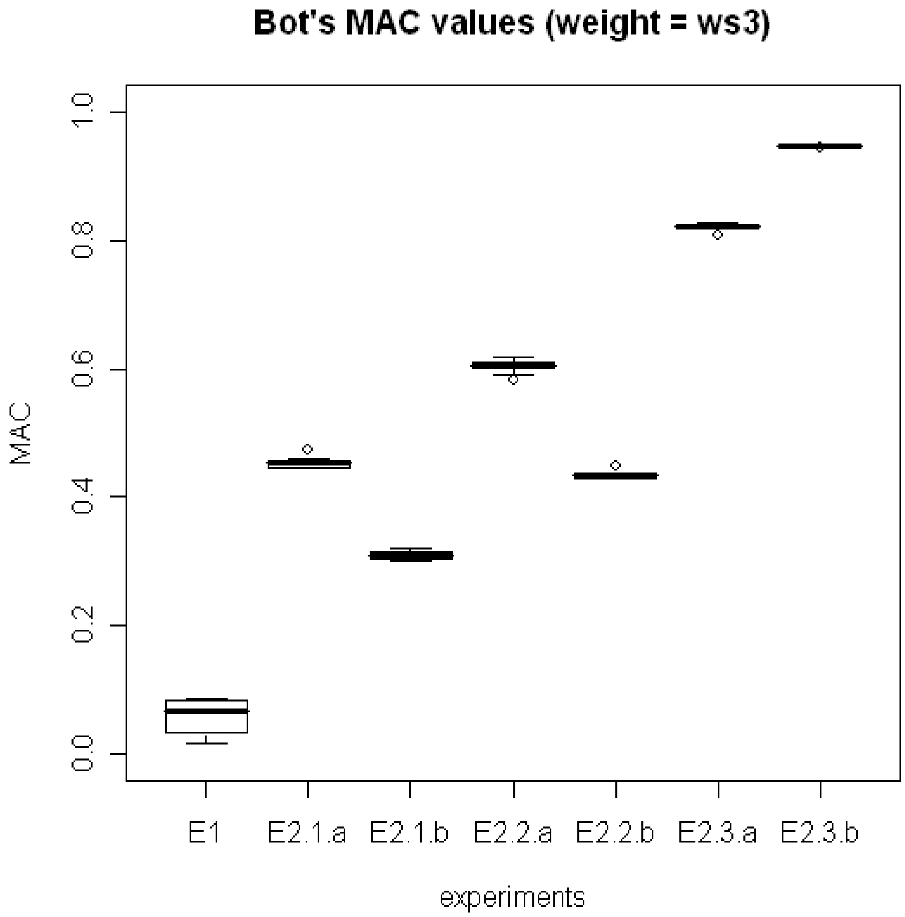

FIGURE 6.4: Bot's MAC values generated by DCA using signal weight WS_3.

has low activity and therefore does not generate any great variation in the signals. Therefore, without input variation, the output does not vary in response to changing the manner in which the input is processed. This is also evident in Figure 6.7 when using the MAC value.

For all other sessions, much greater variation is observed upon weight modification, as shown by the large interquartile ranges produced for both MCAV and MAC values of the bot processes. While similar trends are shown across the sessions in the MCAV of the IRC client, differences are evident for the MAC value. In Figure 6.9 it is evident that all sessions have low MACs for this process across all weight sets. Therefore as

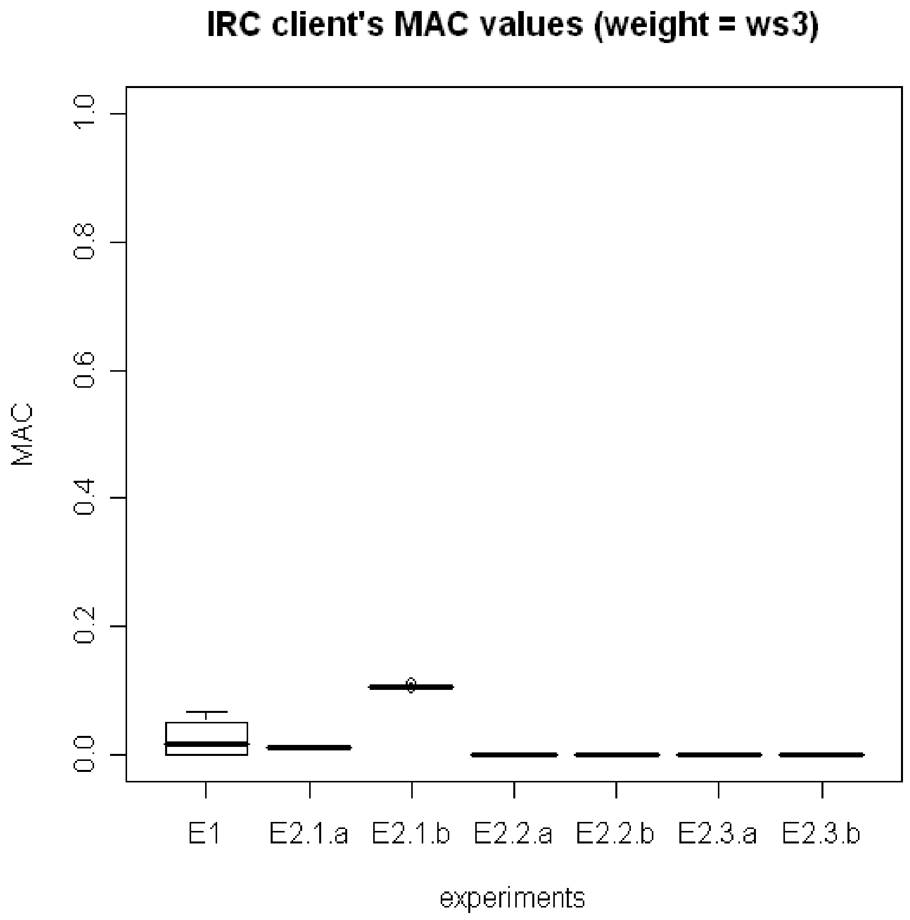

FIGURE 6.5: IRC client's MAC values generated by DCA using signal weight WS_3.

the weights are modified, there is a greater influence on the anomalous processes than on the normal processes. Should the arbitrary threshold applied to the MAC values be set at 0.2 as opposed to 0.5, then the performance of the DCA on botnet detection is good, producing low rates of false positives and high rates of true positives.

To further explore this effect, an alternative plotting is presented in Figure 6.10, Figure 6.11, 6.12 and 6.13. Here, each bar represents the results for each WS, derived from the ten runs per session (E1-E2 inclusive) totaling 70 runs per bar. As with Figure 6.6, Figure 6.7, Figure 6.8 and 6.9 some influence is shown through weight modification.

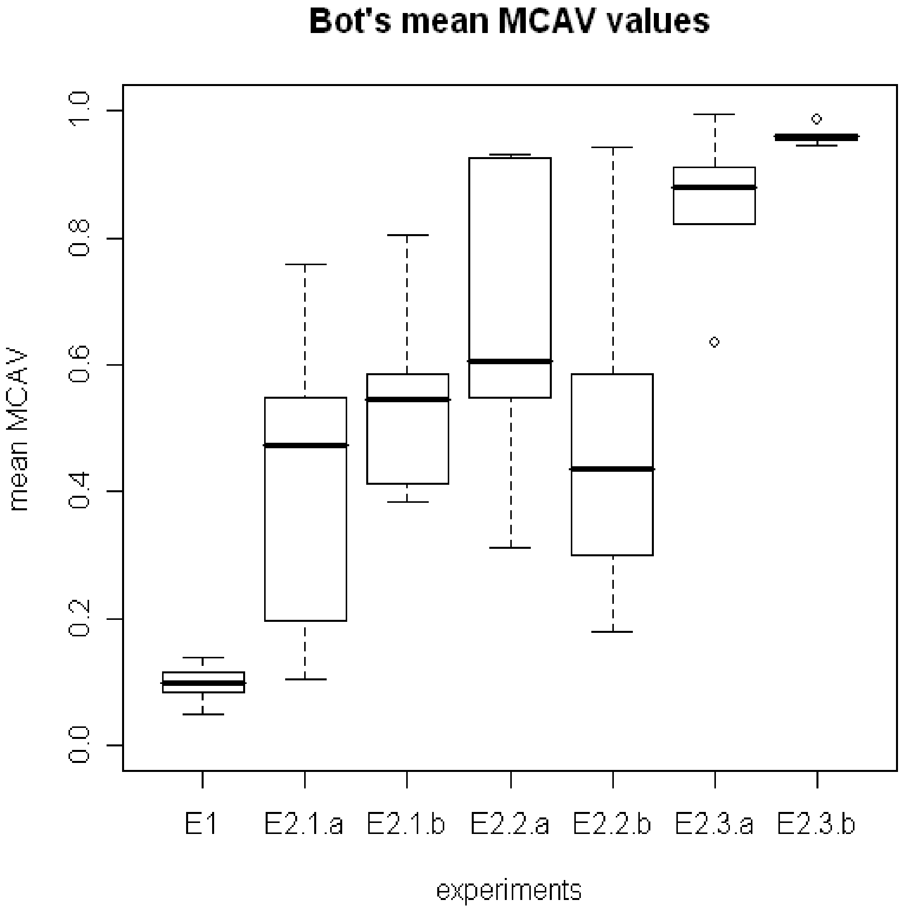

FIGURE 6.6: Bot's mean MCAV values generated by DCA using different signal weight values (WS1-WS5).

WS_1 produced low MCAV and MAC values for both processes. This indicates that these weights, used previously with the DCA, are less suitable for this particular application. WS2 shows little improvement when compared to WS_1, producing the lowest MCAV and MAC for the bot process. WS_3 shows an improved performance, producing much higher MAC values for the bot process and very low values for the IRC client. WS_4 produced even higher values for the bot process, whilst keeping the values low for the normal IRC client process. WS_5 also produced high MCAV and MAC values for the bot, but the interquartile range of the normal process increased. This suggests that as the ratio of PAMP to safe signal weight for producing the mature

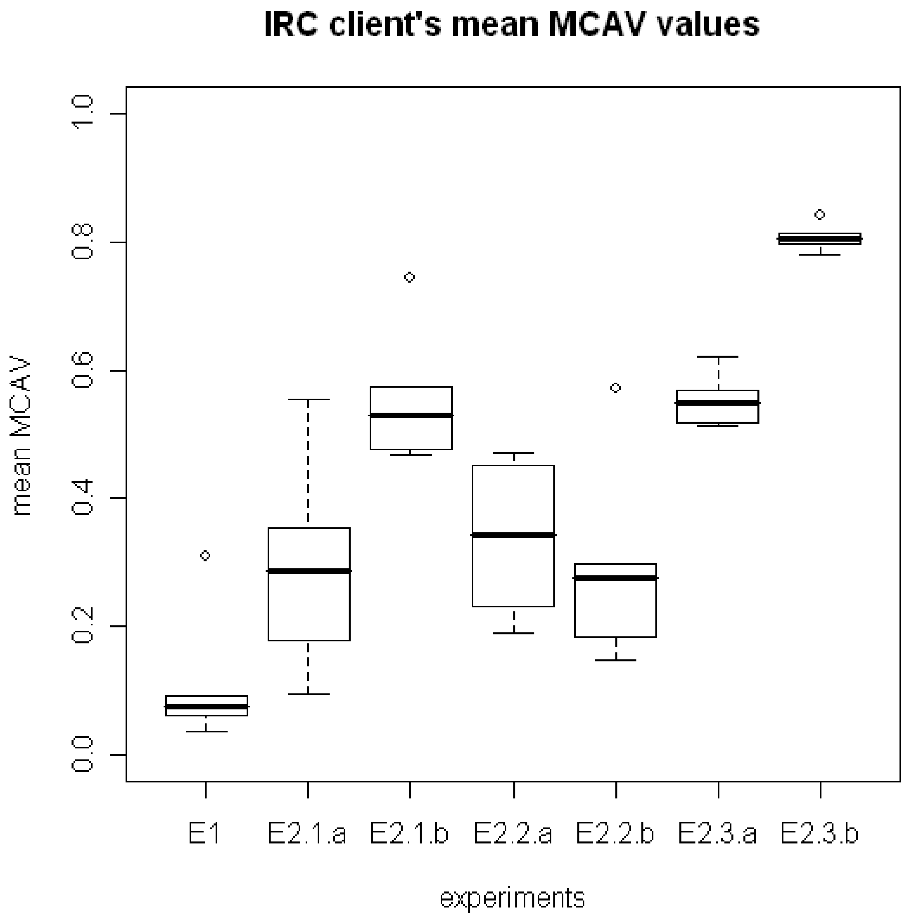

FIGURE 6.7: IRC client's mean MCAV values generated by DCA using different signal weight values (WS1-WS5).

output signal must be sufficiently small to avoid false positives, and sufficiently large to avoid potential false negatives.

Finally, to verify these findings statistically, each set of results per session per weight are compared exhaustively using the non-parametric Wilcoxon signed rank test. For each test performed the resultant p-value is less than 0.001. This allows us to conclude that modification of the weights has a significant effect on the output of the DCA when applied to this detection problem, and leads to the rejection of Null Hypothesis Four (H4).

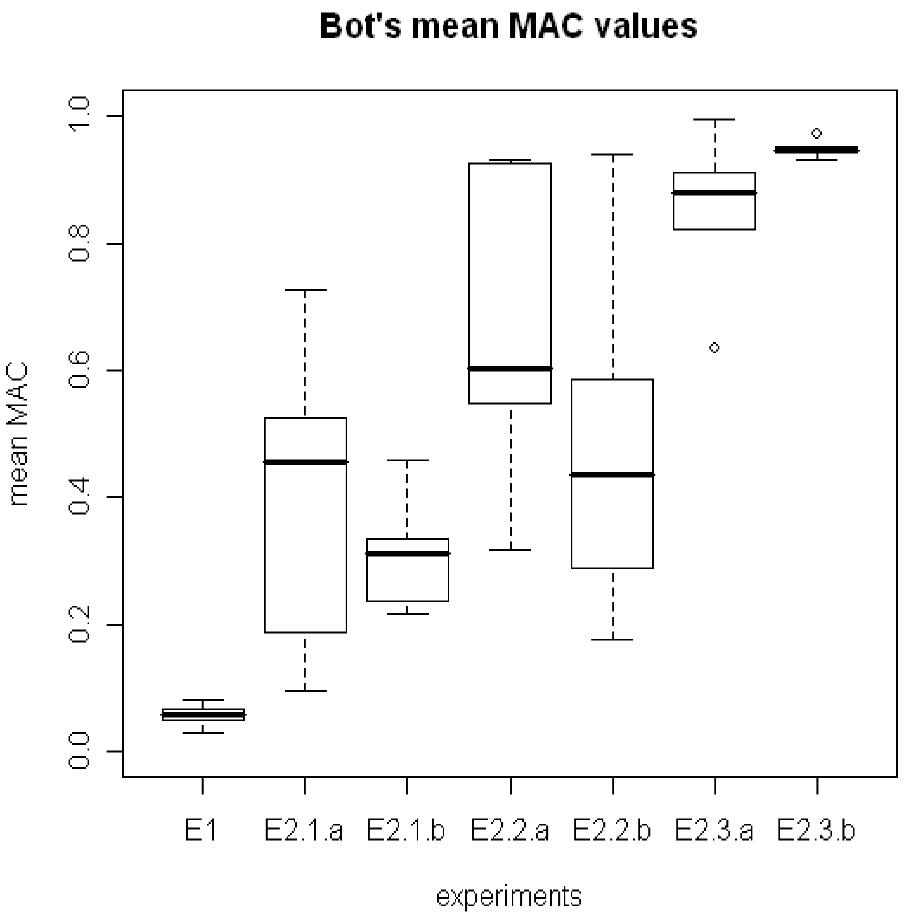

Figure 6.8: Bot's mean MAC values generated by DCA using different signal weight values (WS1-WS5).

Table 6.5: Weight sensitivity analysis for the bot's MCAV values

Expriment	WS1	WS2	WS3	WS4	WS5
E1	0.0484	0.0834	0.0978	0.1140	0.1377
E2.1.a	0.1030	0.1964	0.4736	0.5477	0.7595
E2.1.b	0.3823	0.4123	0.5440	0.5861	0.8032
E2.2.a	0.5488	0.3119	0.6047	0.9269	0.9319
E2.2.b	0.2995	0.1770	0.4360	0.5863	0.9427
E2.3.a	0.8802	0.6345	0.8218	0.9112	0.9955
E2.3.b	0.9553	0.9443	0.9598	0.9641	0.9873

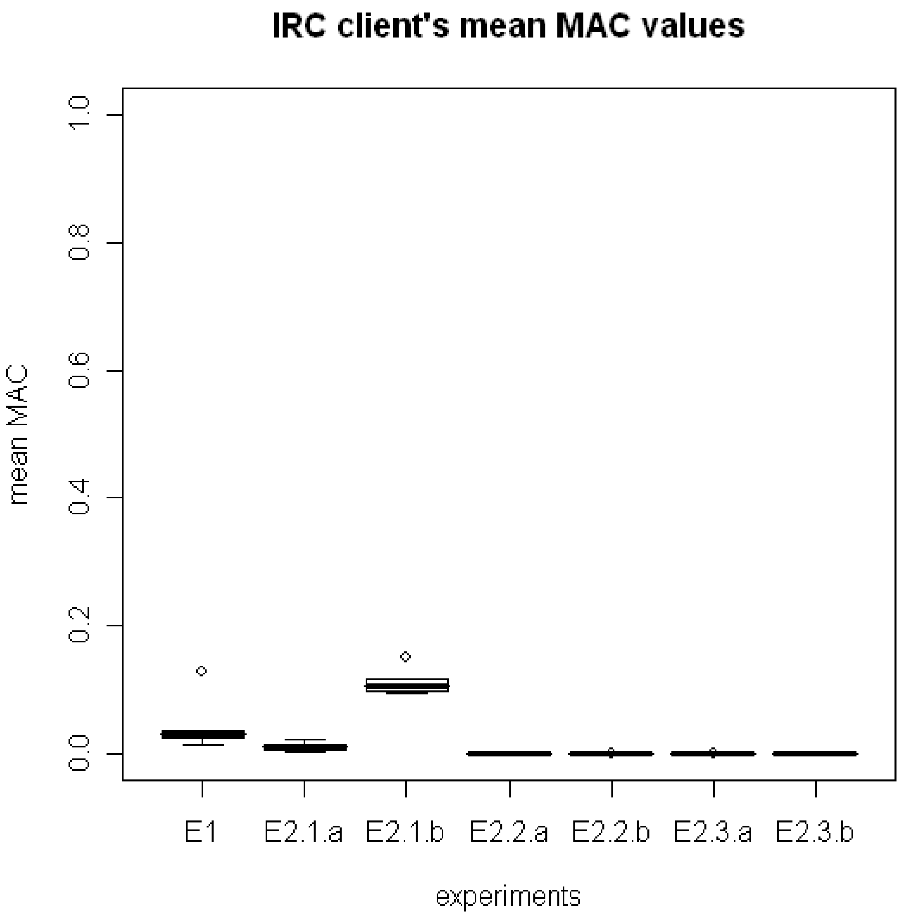

Figure 6.9: IRC client's mean MAC values generated by DCA using different signal weight values (WS1-WS5).

Table 6.6: Weight sensitivity analysis for the bot's MAC values

Expriment	WS1	WS2	WS3	WS4	WS5
E1	0.0288	0.0495	0.0578	0.0671	0.0810
E2.1.a	0.0947	0.1882	0.4543	0.5246	0.7274
E2.1.b	0.2168	0.2355	0.3098	0.3340	0.4572
E2.2.a	0.5479	0.3155	0.6038	0.9255	0.9305
E2.2.b	0.2886	0.1764	0.4345	0.5845	0.9395
E2.3.a	0.8798	0.6342	0.8214	0.9108	0.9949
E2.3.b	0.9418	0.9306	0.9461	0.9507	0.9726

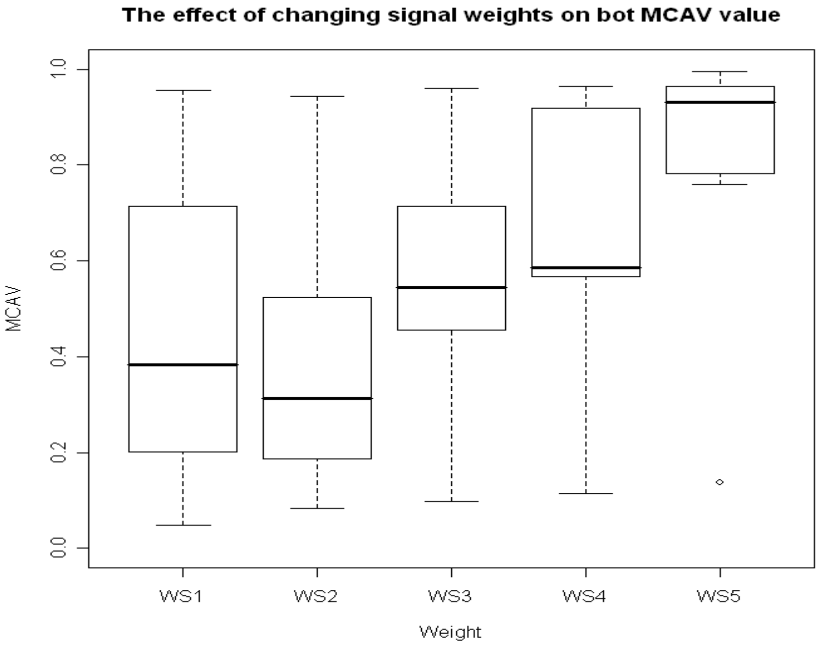

Figure 6.10: Affect of changing signal weights of bot's MCAV values on DCA detection performance.

6.7.3 Spearman's Rank Correlation (SRC) Algorithm and the DCA Performance

From the results obtained, even though that both algorithms were able to detect the malicious behaviours by correlating different attributes, we have noticed that the DCA has a better detection performance over the SRC algorithm when detecting the bot by reducing the number of false alarms. Therefore, the Null Hypothesis Five (H5) can be rejected.

6.8 Conclusions

In this chapter, we present two different correlation algorithms for detecting an individual bot on the system. We first use the Spearman's rank correlation algorithm to correlate different bot's behaviours to enhance the detection mechanism. Even though the Spearman's rank correlation algorithm is a simple way of correlating different behaviours, we have noticed that it detects the bot in most of the cases. One

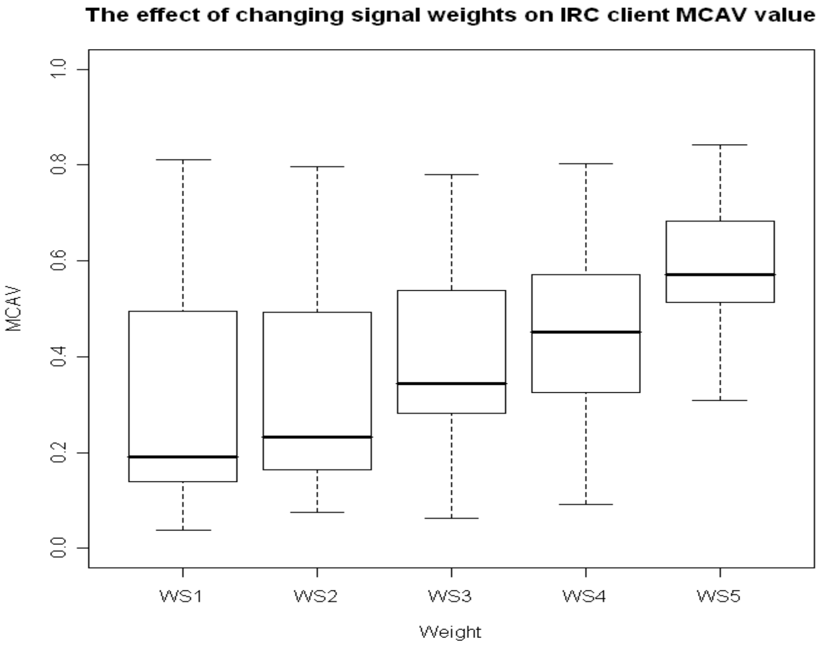

FIGURE 6.11: Affect of changing signal weights of IRC client's MCAV values on DCA detection performance.

problem with the Spearman's rank correlation algorithm was the existence of the idle periods which increases the rate of false negatives. In order to improve the detection mechanism, we have removed these idle periods and compared the new results with the previous results. We have noticed that when removing the idle periods, the Spearman's rank correlation algorithm gives a better detection rate. The second problem with the Spearman's rank correlation algorithm is that for normal scenarios, the algorithm classifies the normal applications as anomalous by generating a high correlation value. This is due the fact that there is an increase of idle periods. In order to improve the correlation algorithm, a more intelligent approach is needed, therefore, the Dendritic Cell Algorithm (DCA) is used.

The DCA is an artificial immune system which is based on the biological immune system to detect foreign antigen, in our case the bot, which are not part of the environment. Different signals are used with the combination of the antigen. The signals are the behaviours of the bot while the antigen are the API function calls. We

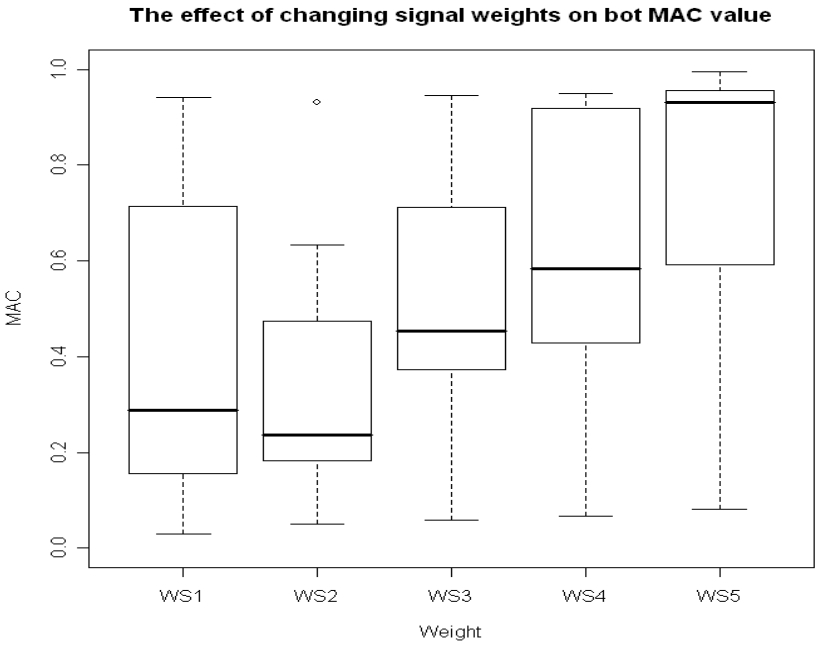

FIGURE 6.12: Affect of changing signal weights of bot's MAC values on DCA detection performance.

have used the DCA to detect the existence of an individual bot by correlating the signals with the antigen within a specified time window. The results show that the DCA performs better than the Spearman's rank correlation for detecting normal and abnormal behaviours. In addition, the DCA was able to detect abnormal processes with few false alarms compared to the Spearman's rank correlation.

Additionally, the incorporation of the MAC value has a significantly positive effect on the results, significantly reducing false alarms. Finally, the modification of the weights used in the signal processing component has a significant effect on the results of the algorithm. It is concluded that appropriate weights for this application include high values for the safe signal (SS) weight which appears to be useful in the reduction of potential false alarms without generating false negative errors.

We have seen that there are some values which are determined by experimentation. For example, the critical values of S_2 and S_3 and their ranges were selected to represent different levels of the threat. Although in our experimentation we tried to provide

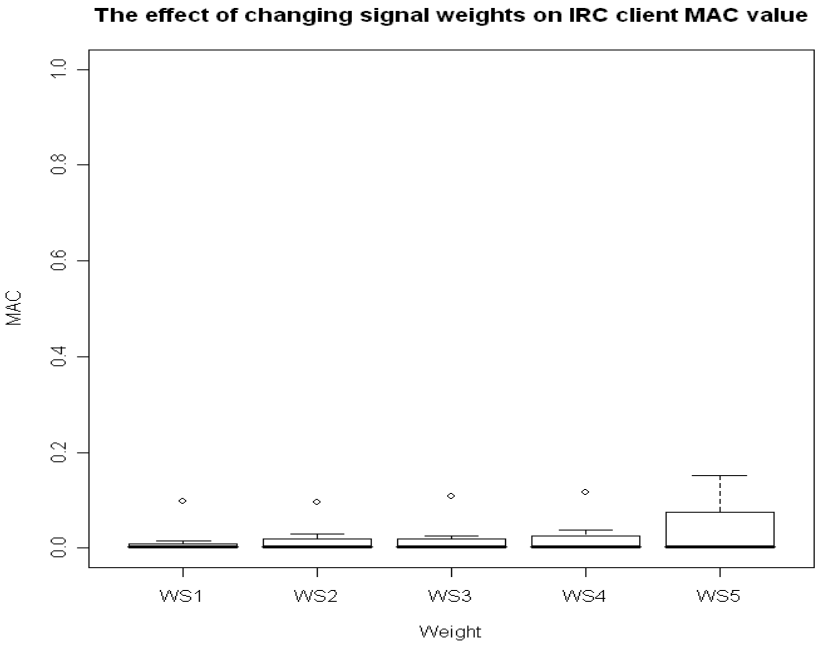

Figure 6.13: Affect of changing signal weights of IRC client's MAC values on DCA detection performance.

proper ranges and values to detect different types of bots, these ranges can be defeated if the botmaster programmed his bots to respond in a very slow manner, thus allowing the DCA to generate false alarms. In such a case, we expect that bots will loose some of their purpose they are made for, such as to provide fast response before detection and participating in DDoS by driving large amount of traffic to target other systems within a short period of time.

For the computational complexity of the DCA, Feng et al. [56] showed that the DCA has a runtime complexity of a lower bound of (n) and an upper bound of O(n2). He also suggested that this complexity can be improved to O(nlogn) by using segmentation.

In the next chapter, we intend to apply the DCA to the detection of "peer-to-peer" bots, which pose an interesting problem as the use of peer-to-peer networks increases. In the methodology chapter, we have stated that using multiple signals from the same signal category can provide flexibility to our system and reduce the effect of a proper selection of one signal type. In the next chapter, we will examine the case where we have multiple PAMP signals to produce an average PAMP value. In addition we aim to use the results of these experiments to further our understanding of the DCA, to ultimately enhance the performance of this immune-inspired detection algorithm.

Chapter 7

Host-Based Detection for Peer to Peer (P2P) Bots using DCA

7.1 Introduction

Recently, a new approach of botnet structures started to appear taking advantage of existing Peer-to-Peer (P2P) protocols. Many attackers started to use P2P networks in order to control their botnets. By using this approach, the botmaster can contact other bots without having a centralised point for their command and control (C&C) structure. When using a centralized structure, one can prevent the bots from communicating with their masters by shutting down the central point. In order to avoid this problem, botnet herders started to deviate from using a centralised point, such as IRC structures, to a more efficient way of controlling their botnets by using the P2P protocols as a mean to maintain their botnets. In P2P, each node acts as a client-server which provides bandwidth, storage and computational power. Using this approach, bots are able to communicate with other bots by downloading files or commands from other bots' machines and performing different activities. In comparison to IRC structures, everyone can join a P2P network, thus, the more peers acting as bots, the more powerful the botmaster can be. In addition, it will be hard to detect and shut down the botnet as security people would need to isolate each machine [37].

Because our main focus is to detect an individual bot on a machine, we will use the Dendritic Cell Algorithm (DCA) with pre-defined signals in order to detect the bot. These signals are derived from monitoring the behaviour of the P2P bots. We will use two bots (Phatbot and Peacomm) in our detection experiments as a case study. In the case of Phatbot, this is the only P2P bot that we found with its source code and modified it according to our needs.

This chapter is structured as follows. We start by giving a brief history of P2P protocols as in Section 7.2 and explain two case studies of P2P bots. In Section 7.3, we apply the DCA to detect the P2P bots in which we present the bot scenarios, how we collected our data, the experiments that we have conducted and different null hypotheses that are used for this investigation and finally analyse and discuss our results. In Section 7.4, we evaluate our algorithm with existing bots detection algorithms and develop a non-DCA algorithm for further evaluation. We conclude with a discussion of the results.

7.2 Background and History

Peer-to-Peer (P2P) protocols were used previously mainly for distributing files among other peers over the network. The first generation of P2P networks were based on connecting to a main server, querying for specific files and the main server responds to the query about the location of that file. Then, the peer connects to the specified peer and downloads the requested file. This method is used by Napster for file sharing as shown in Figure 7.1. In Napster [43], the following steps are required in order to connect to other peers and download the file:

- the client connects to the Napster server.
- the server accepts the connection.
- the client sends a query describing the requested file to download.
- the server processes the query and sends a response to the client containing the locating of the host that has the file back to the client.
- the client disconnects from the server.

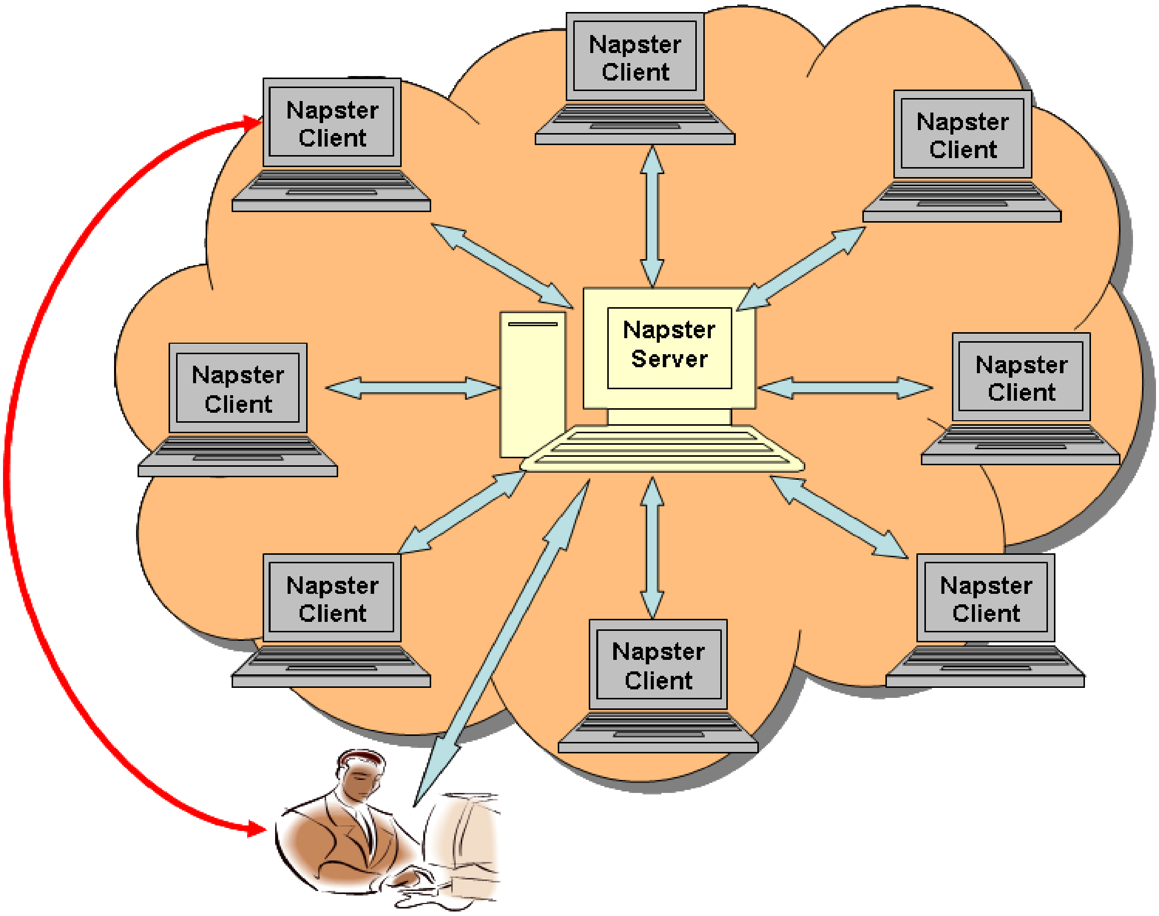

FIGURE 7.1: P2P structure:napster

- the client tries to connect to the host(s) that have the file.
- Once the connection is established, the client requests the file.
- the connected host responds to the request.
- the client downloads the requested file and disconnects from the other host.

In contrast, Gnutella [43] has a pure P2P structure in which there is no central server to connect to. In addition, there is no central database which knows the location of all files. The peer forwards its query to its neighbour peers. The host in a P2P network is called a servant which acts as a server and client simultaneously. The process of finding a file in a Gnutella network works as shown in Figure 7.2.

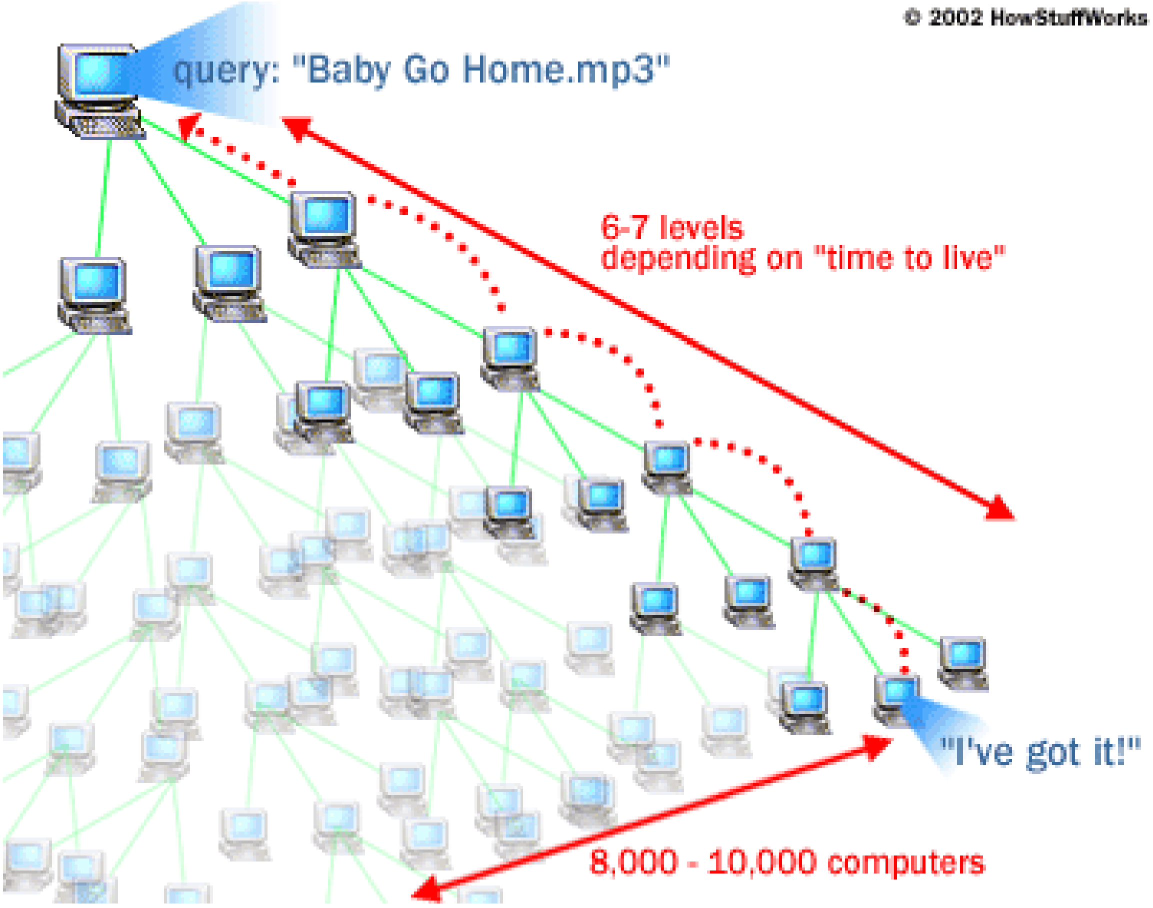

FIGURE 7.2: P2P structure:Gnutella (source:http://www.howstuffworks.com)

- the servant connects to its neighbor and sends a query searching for a specific file.
- the neighbour hosts check if they have the file.
- if one of the neighbours has the file, it responds to the servant.
- otherwise, every neighbour connects to its neighbour and sends the query by repeating the same process and so on.
- after a certain time, the querying process is stopped due to Time-to-Live (TTL) expiry.

- if the servant receives response in this process, it connects to the host storing the file.
- the host storing the file accepts the connection and responds to the servant.

Because P2P networks are hard to shut down, botmasters start to incorporate existing P2P protocols into bot code. Other botmasters have developed their own P2P protocols in order to control their bots. In the next section, we will explain the two case studies that we will use in our P2P detection experiments.

7.2.1 Case Study I: Phatbot

Phatbot is derived from the Agobot family. Phatbot has embedded the P2P structure in its code to control it instead of using IRC structures. Phatbot uses Gnutella cache servers for its bootstrap process to find other peers by registering itself with a list of URLs using GNUT, a Gnutella client [171][143]. In order to communicate with other infected hosts, each host connects to these cache servers to retrieve the list of Gnutella clients. The author of the Phatbot uses a P2P protocol called WASTE, a project created by AOL's Nullsoft division [172] for small networks. The author of Phatbot removes the encryption from the WASTE code to delete the process of sharing of public keys. In order to control the Phatbot WASTE network, a WASTE client is needed to connect to a peer found on the cache servers. One problem when using WASTE clients is that the WASTE protocol is designed for small networks, therefore, the scalability issues are raised [131][143].

7.2.2 Case Study II: Peacomm (Storm Bot)

Peacomm generated in 2007, is one of the few known bots which implements a full peer-to-peer (P2P) networks of bots [116][57] which uses the Overnet P2P protocol to communicate with other bots. The overnet protocol is an example of a decentralised peer network which implements the Kademlia algorithm [116]. The Kademlia algorithm uses distributed hash tables (DHT) for routing in which there is no hierarchy in the topology [57][116].

In Kademlia, to communicate with other peers, each node is assigned a unique 128-bit ID as a global identifier the first time the it starts [71]. The node shares its information with other peers which are close to it in the keyspace. The keyspace contains for example, 128-bit strings where every peer in the P2P network share some portion of that keyspace and the near ones were they can find other peers using a hash function. The values are mapped into the string space with keys. Key/value pairs are stored on the "closest" nodes. The term "closeness" is based on calculating the distance between two peers' identifications (IDs) using an XOR operation [103][71]. To publish information, each node uses its hash table as a data structure which maps the keys with the values. The key is used as an identification to retrieve information based on the closest distance to other peers while the value is a triple of node's ID, IP address and UDP port number. The Peacomm bot uses this method to search for specific keys to communicate with other infected machines on the network and find the commands that should be executed [71].

The life cycle of a Peacomm bot involves: (1) infection, (2) installation of initial bot in the infected host and (3) downloading of the secondary injection into the host. After an infection, the bot publishes itself on the Overnet network and starts to connect to peers using the initial list of peers that is hard coded in the bot [71]. If these connection attempts failed for some reason, we will see large number of destination unreachable or connection resets which are used as our PAMP signal. On the other hand, each time the bot makes a successful connection, it expands its information about the botnet by saving the hash (node information) into local memory. The bot then uses the stored hard coded keys to search and download a secondary injection value which is an encrypted URL that points to the location of a secondary injection executable (code) from the P2P network. Accourding to Stewart [145], the search key for secondary injection is generated using implemented algorithm which uses the current date and a random number from [0..31] as input to the algorithm. The bot decrypts the downloaded encrypted URL using its stored hard coded keys. The bot then downloads the secondary injected executable from a web server using the decrypted URL. This executable may contain bot's command to perform malicious activities such as DDoS, spam, install rootkits or spread. Note that no explicit delivery of commands is used in this process. Various secondary injection executables are used such as downloader and rootkit component, SMTP email spamming component, email address harvester for the previous spamming stage, email propagation component and distributed denial of service tool [116][57].

7.3 DCA for detecting P2P Bots

7.3.1 Introduction

In chapter 6, we have seen that the Dendritic Cell Algorithm (DCA) can successfully detect IRC bots. In this section, we extend our investigation to include peer-to-peer (P2P) bots. We apply the DCA to detect the two case studies bots (Phatbot and Peacomm). The aim of this investigation is to show that the DCA can detect bots which uses different C&C communication protocols.

We start by investigating the performance of DCA on detecting Phatbot as we already have the source code for this bot, thus controlling the bot to do malicious activities is possible. Different scenarios are conducted to achieve this task with different null hypotheses. Then we apply the DCA on the Peacomm bot by running the binary because the source code for this bot is not available. The results show that the DCA classifies the two bots as anomalous processes.

7.3.2 Methodology

Bot Scenarios

For the purpose of our experimentation two case studies of P2P bots (Phatbot and Peacomm bot) are used as previously explained. As we have the source code of the Phatbot, we are able to control this bot to do malicious activities such as information gathering and flooding attack. In the case of Peacomm bot, we did not manage to get the source code. As a communication vessel, IceChat [77], an IRC client, is used for normal conversation. A WASTE client [172] is used for both controlling the bot as well as a normal application. In addition, the Firefox [36] web browser is used for browsing, checking email and other normal activities as another normal application.

For data collection, an APITrace intercepting program is used to record the required behavioural attributes and to intercept and capture function calls. For the Phatbot experiments, three different scenarios are constructed including idle (PhatE1), attack (PhatE2) and normal (PhatE3) scenarios. The attack scenario consists of two sessions: information gathering session, and a flooding session.

- *idle bot (PhatE1):* This session involves having an *idle Phatbot* running on the monitored host in addition to normal applications such as an IRC client, the WASTE client and the Firefox web browser. The Phatbot runs on the monitored victim's host and listens for an incoming connection from the attacker. Once it connects to the WASTE client, the bot then joins a specified channel waiting for commands from its controller, though no attacking actions are performed by this bot.

- *Attack (PhatE2):* two sessions are used for this stage. Once the Phatbot connects to the WASTE client and joins the channel, the attacker starts to issue different commands but does not include flooding commands. These commands include obtaining information from a victim's machine, retrieving commands, monitoring the user's activities on the system, opening and deleting files which represents session *PhatE2.1.* The second session, *PhatE2.2*, involves performing packet flooding using a SYN flood attack, UDP attack and ICMP attack. These flooding methods are designed to emulate the behaviour of a machine partaking in a distributed denial of service attack. As part of the process of packet flooding the bots rely heavily on socket usage, as part of the packet sending mechanism. Therefore to detect these attacks, socket usage monitors are employed, with the exact nature of this data given in the forthcoming section. Note that during the flooding attack other 'normal' legitimate applications are still running.

- *Normal Scenario (PhatE3):* The normal scenario involves browsing the net through Firefox, and chatting through the WASTE and Icechat clients. The APITrace intercepting program is running on the victim's host to collect the data. Note that no bots are used in this scenario as this is the normal/uninfected session used to assess the occurrence of any potential false positive errors made by the DCA.

For the Peacomm bot, two scenarios are used inactive (PmE1) and active (PmE2) as follows:

- *inactive (PmE1):* In this session, the binary of Peacomm bot is executed and runs on a monitored host. Other normal applications are also running during this session but there are no activities from the user such as browsing or chatting.
- *active (PmE2):* In this session, the Peacomm bot is executed and runs on a monitored host. In contrast to (*PmE1*), the user uses Firefox for browsing and checking emails as normal activities and uses Icechat for having conversation with other users. In both cases, the APITrace is running to collect the data.

Data Collection

The same assumptions as when using DCA to detect IRC bots are used for the P2P bots. The two bots are already installed on the victim's host, through an accidental 'trojan horse' style infection mechanism or opening a malicious attachment which contains the bot. In this case, we use an extrusion detection to limit the bot activities whilst on a host machine. The Phatbot runs as a process whenever the user reboots the system and listens for incoming connections from the attacker. The communication of Peacomm bot is described in more details in [71][57][152].

An interception program (APITrace) is used to collect the required data which is processed, normalised and streamed to the DCA. In terms of the function calls intercepted, different types of function calls are used as an input to the DCA. These function calls include Communication functions, File access functions, Registry access functions and Keyboard status functions.

Signals

Signals are mapped as a reflection of the state of the victim's host. A major function of the DC is the ability to combine signals of different categories to influence the behaviour of the artificial cells. Three signal categories are used to define the state of the system namely PAMPs, DSs (danger signals) and SSs (safe signals), with one data source mapped per signal category. These signals are collected using a function call interception program - APITrace. Raw data from the monitored host is transformed into log files which are then analysed by the DCA, following a signal normalisation process. The resultant normalised signals are in the range of 0 - 100 for the PAMP, danger signal (DS) and safe signal (SS).

In terms of the signal category semantics, the PAMP signal is a strong signal evidence for bad behaviour on a system. This signal is derived from the rate of change of three fields of *netstat*. These fields are destination unreachable (DU), failed connection attempts (FCA) and reset connections (RST). The choice of these fields is based on the preliminary observation of P2P bots. The *netstat* is a command line tool used to display network statistics. The value of PAMP signal is obtained from the following formula:

$$PAMP_t = (DU_t - DU_{t-1}) + (FCA_t - FCA_{t-1}) + (RST_t - RST_{t-1}) \qquad \forall t$$

The normalisation of data is based on a logarithmic scale. This is because we needed to cover data of large range of values being produced by the flooding attack and these values are changing rapidly. If the value of the PAMP signal exceeds 100, the value is capped to a maximum value, in our case 100. Otherwise, the value of PAMP signal is calculated as shown in Table 7.1.

The danger signal is derived from the rate of change of number of packets sent per second (pkts/sec). This value is also obtained from network statistics command line tool (netstat). Based on preliminary experiments, we classify our danger ranges (i.e. number of packets sent per second [pkts/sec]) according to the following:

Table 7.1: Values of PAMP signal for P2P experiments

DU	RST	FCA	PAMP
0	0	0	0
0	0	!=0	100*log10(RST)
0	!=0	0	100*log10(FCA)
!=0	0	0	100*log10(DU)
0	!=0	!=0	100*log10(RST+FCA)
!=0	0	!=0	100*log10(DU+FCA)
!=0	!=0	0	100*log10(DU+RST)
!=0	!=0	!=0	100*log10(DU+RST+FCA)

$$X = \begin{cases} 0 - 10 & \textit{min danger} \\ 11 - 100 & \textit{min-mid danger} \\ 101 - 1000 & \textit{mid danger} \\ 1001 - 10000 & \textit{mid-max danger} \\ > 10000 & \textit{max danger} \end{cases}$$

We also use a logarithmic scale when using the danger signal. The danger signal value is derived according to the following formula:

$$DS = 25 * log10(X) \ldots 1 \leq X \leq 10,000$$

If the rate of change of the number of packets sent per second exceeds 10,000, the value of DS is capped to 100. In addition, if the rate of change is zero, the value of DS is mapped to zero.

Finally, the safe signal is derived from the time difference between two outgoing consecutive communication functions such as [(send,send),(sendto,sendto),(socket,socket)]. This is needed as the bot either sends information to the botmaster using *send* function call or issues SYN or UDP flooding attacks using *sendto* or *socket*. In normal situations, we expect to have a large time difference between two consecutive functions. In addition, we expect to have a short period of this action in comparison to a SYN attack or a UDP attack. This is because the behaviour of the user when responding to other parties is different from bots when responding to their botmaster's commands. Often, the normal users do not involve in generating large amount of traffic similar to the flooding attack when they chat or when they browse the Internet. Therefore, we set n_s as the maximum time difference between calling two consecutive communication functions. If the time difference is higher than n_s, the time is classified as normal and it is mapped to 100. If the time difference falls below n_s, the time difference is calculated from the following formula:

$$SS = 62.41965 * log10(Y)$$

The value of 62.41965 is calculated from preliminary experiments in which we observe that the *average safe* value is around 40 and any value above this value can be considered to be as normal. Therefore, if $Y = 40$, the value of SS is capped to 100 which is the *max safe* value. The closer the value of SS to 0, the more *min safe* it is.

This approach of detecting P2P bots using the DCA is similar to detecting IRC bots with the difference being the types of signals being used. In comparison to detecting IRC bots, P2P PAMP signal differs from the IRC PAMP signal due to the nature of P2P bots. We added different PAMP signals to form one PAMP signal. These PAMP signals include DU, RST, FCA and keylogging activity. Because we did not find P2P bots that perform keylogging activity, the value for keylogging activity is assigned to zero. For the danger signal, using IRC bots, we wanted to examine the behaviour of bots when receiving commands in comparison to human behaiour. We noticed that the bot responds in a very fast way to the botmaster commands. This activity directs us to use how fast the bot responds to the attacker's commands as our danger signal. Although in P2P bots we could use the same danger signal similar to IRC bots, but we noticed that the amount of traffic being generated when monitoring P2P bots is very large which has a strong indication of malicious activity which we considered it as our danger signal. The safe signal remains the same for both IRC bots and P2P bots.

The selection of these signals depends of the applications. This makes the DCA applicable for different application areas, thus providing flexibility and scalability. In addition, different categories of signals can be used to form one major signal. If one signal is not selected properly, the other signals within the same subset can substitute the missing signal. For example, we could combine the danger signals of IRC bots and P2P bots and take the mean value to form the main danger signal. We consider the flexibility of signal selections as one of the features of the DCA in which it can adapt to many applications using different signals. The choice of, which is the best signal to consider can be determined by the end user.

Antigen

Antigen represents potential culprits that are responsible for changes in the status of the monitored system. The need for correlation between antigen and signals is required to define which processes are active when the signal values are modified. The more active the process, the more antigen it generates. Once the function calls are intercepted by APITrace, they are stored and assigned the value of the process ID to which the function calls belong and the time at which they were invoked. After a certain period of time, both signal and antigen logs are combined and sorted based on time. The combined log files are parsed and the logged information is sent to the DCA for processing and analysis. An example of this file is provided in the index.

7.3.3 Experiments

The aim of these experiments is to measure the performance of the DCA on detecting P2P bots. Each experiment is repeated ten times which produces a small variation of PAMP, DS and SS on standard deviation using Chebyshev's Inequality. One dataset is selected randomly from each repeated experiment and passed to the DCA using a data collection client. Each scenario is used for the purpose of experimentation with three hypotheses tested:

1. *Null Hypothesis One (H1):* Data collected per dataset are normally distributed. The Shaprio-Wilk test is used for this assessment.
2. *Null Hypothesis Two (H2):* The MCAV/MAC values for the benign application programs are not statistically different from those produced by the P2P bots.

This is verified through the performance of a two-sided Mann-Whitney test, where p value is equal to 0.95.

3. *Null Hypothesis Three (H3):* The numbers of false positive alarms for the MCAV and the MAC values are the same.

7.3.4 Design and Implementation

The parameters used in DCA are identical to those implemented in Greensmith and Aickelin [52], with the exception of the weights. In our case, we use weight signal WS_3 from Table 6.1 as it generated the best results when applying the DCA to IRC bots. All experiments are performed using a Windows XP P4 with 2.4 GHz processor. The statistical analyses are performed using R statistical computing package (v.2.6.0).

7.3.5 Results and Analysis

After collecting the required data (signals and antigen), the data is passed to DCA to see if it can distinguish between normal processes and malicious processes.

Null Hypothesis One (H1)

By using the Shapiro-Wilk test to each of the datasets, the resultant p-values imply that the distribution of scenarios PmE1 and PmE2 are normally distributed for DS and SS, with PhatE1, PhatE2.1, PhatE2.2 and PhatE3 not normal. Given these two rejections, Null Hypothesis One (H1) is rejected. As a result of this, further tests with these data use non-parametric statistical tests such as the Mann-Whitney test, also using 95% confidence.

Null Hypothesis Two (H2)

In experiment PhatE3, a normal scenario is used in which we monitor the behaviour of a clean system. The DCA did not produce any malicious activities running on the system. This is noticed by generating low values for both MCAV and MAC.

TABLE 7.2: The results of the MCAV/MAC values generated from DCA using signal weight WS_3

Experiment	Process	Processed Antigen	mean		Mann-Whitney P-Value	
			MCAV	MAC	MCAV	MAC
PhatE1	Phatbot	4362.3	0.4949	0.2065		
	Firefox	3524.9	0.3312	0.1115	0.00018	0.00018
	Icechat	1846.9	0.1621	0.0285	0.00001	0.00018
	WASTE	178	0.2049	0.0072	0.00001	0.00017
PhatE2.1	Phatbot	5615.5	0.4743	0.1724		
	Firefox	7394.2	0.2082	0.0996	0.00018	0.00018
	Icechat	1870.8	0.2685	0.0324	0.00018	0.00001
	WASTE	147.3	0.2943	0.0026	0.00018	0.00017
PhatE2.2	Phatbot	6630.7	0.5440	0.2201		
	Firefox	6337.1	0.3298	0.1275	0.00018	0.00018
	Icechat	2474.5	0.2032	0.0305	0.00018	0.00018
	WASTE	180.5	0.2836	0.0030	0.00018	0.00017
PhatE3	Phatbot	0	0.0000	0.0000		
	Firefox	4240.1	0.1239	0.0827	N/A	N/A
	Icechat	1630.3	0.0030	0.0007	N/A	N/A
	WASTE	0	0.0000	0.0000	N/A	N/A
PmE1	Peacomm	134985.2	0.5651	0.5554		
	Firefox	2000	0.3737	0.0054	0.00018	0.00016
	Icechat	47.2	0.1067	0.0000	0.00018	0.00006
PmE2	Peacomm	136521.6	0.5014	0.4851		
	Firefox	2695.3	0.3996	0.0076	0.00018	0.00017
	Icechat	1421.9	0.3378	0.0033	0.00018	0.00016

For all other experiments PhatE1, PhatE2.1 and PhatE2.2, we compare the MCAV/MAC values of Phatbot with other benign programs (Firefox, Icechat and WASTE). The same procedure is applied to experiments PmE1 and PmE2. The results of this comparison are presented in Table 7.2. In this table, the computed p-values using an unpaired Mann-Whitney test are presented. Significant differences are shown by the low p-values presented in Table 7.2 for all experiments for both the MAC and MCAV coefficient values, where the sample size is equal to ten. For example, the MCAV value for the Phatbot is higher than the MCAV value of Firefox by at least 50% and by at least 80% when using the MAC value in experiments PhatE1, PhatE2.1 and PhatE2.2. This percentage increases when we compare the MCAV/MAC values with Icechat and WASTE client.

The above information is also displayed in Figure 7.3, Figure 7.4, Figure 7.5 and Figure 7.6 which represent the MCAV for Phatbot, Fireforx, Icechat and WASTE client respectively. In all figures, the y-axis represents the MCAV/MAC values from zero to one and x-axis represents the conducted experiments. Experiment PhatE3 represents a clean system and no bot's activity is detected during this experiment and a low MCAV value is generated for the Firefox and Icechat applications. In experiment PhatE1, no user activity is performed during this experiment but a high MCAV is generated for the bot compared to the MCAV values of benign programs. This is because the bot uses the cache servers to connect to other peer bots. Experiment PhatE2.1 represents the case where the attacker retrieves sensitive information from the infected hosts. The user is active during this experiment. The results show a high MCAV is generated for the bot in comparison to benign programs. In the final experiment, PhatE2.2, the attacker uses the bot to launch a flooding attack and at the same time, the user is active during the attack. The results show that the MCAV for the bot is higher than the MCAV of benign programs. In addition, the MCAV of the bot in this experiment is higher than the previous experiments. For experiments PhatE1, PhatE2.1 and PhatE2.2, if a threshold of 0.4 is set, then the bot will be detected and no false positive alarms will be generated.

In the same procedure as the MCAV, the MAC values of experiments PhatE1, PhatE2.1, PhatE2.2 and PhatE3 for Phatbot, Firefox, Icechat and WASTE client

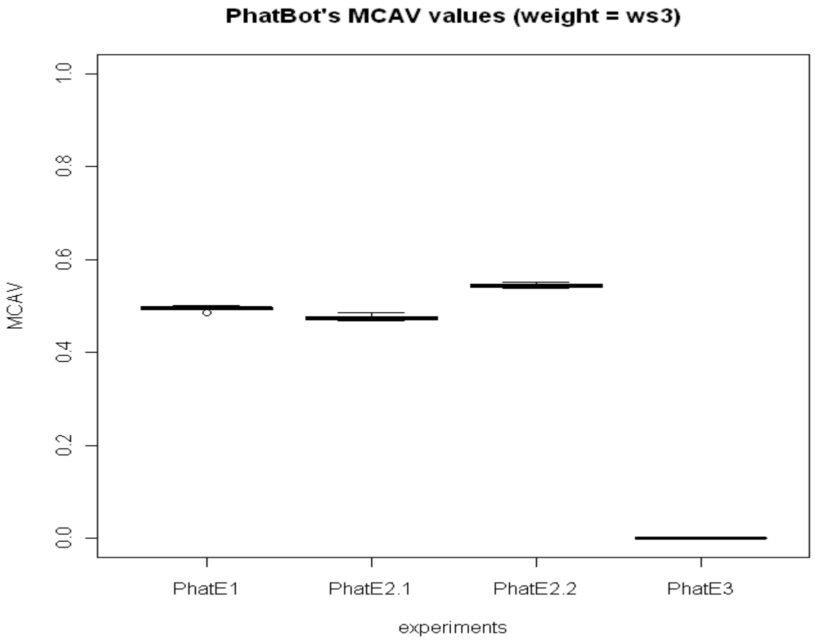

FIGURE 7.3: Phatbot's MCAV generated by DCA using signal weight WS_3.

respectively are displayed in Figure 7.7, Figure 7.8, Figure 7.9 and Figure 7.10. Using the MAC value, it is clearly shown that the bot has a higher MAC value in comparison to other benign programs.

In the case of Peacomm bot, on average the same percentage is obtained as in Phatbot experiments when we compare the MCAV value of Peacomm bot with the MCAV values of benign programs. This information is also displayed in Figures 7.11, 7.12 and 7.13. The y-axis represents the MCAV/MAC values from zero to one and x-axis represents the conducted experiments. The MCAV of the Peacomm bot is higher than the MCAV of benign programs.

The differences are further pronounced in the generation of the MAC values, further supporting its use with the DCA as shown in Figure 7.14, Figure 7.15 and Figure 7.16. The high value of Peacomm's MAC in Figure 7.14 is due to the large number of generated function calls as the bot tries to connect to other peers by sending large amount of UDP traffic.

The effect of changing threshold on MCAV/MAC is shown in Table 7.3 and Ta-

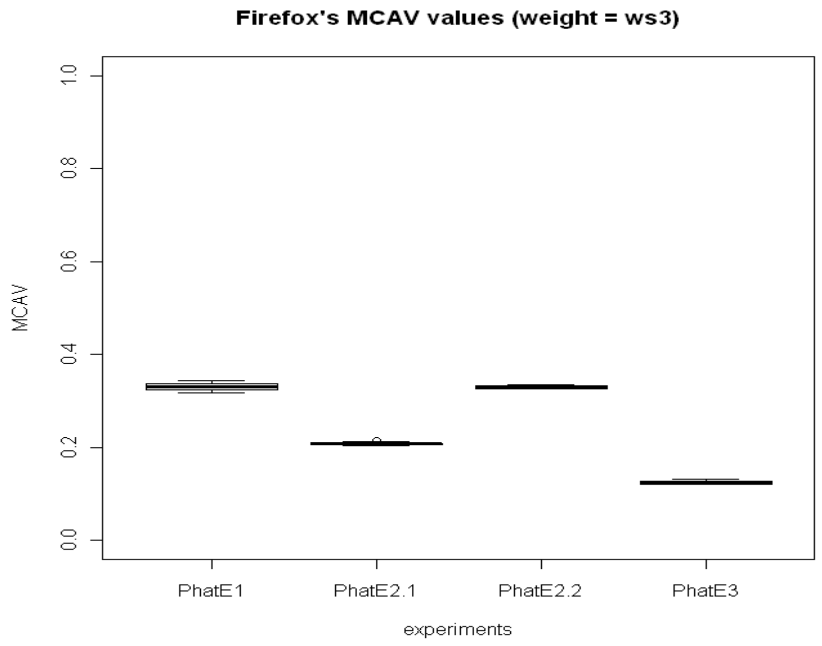

Figure 7.4: Firefox's MCAV generated by DCA using signal weight WS_3.

ble 7.4 respectively. We can see from Table 7.3 that if the threshold is set to 0.4, there will be no false positive values. At the same time, we achieve 100% of true positive values. In Table 7.4, setting a threshold to 0.16 generates zero false positive values and 100% true positive values. The effect of changing the threshold values also can be shown using the ROC analysis for all experiments using the MCAV and MAC values as shown in Figure 7.17 and Figure 7.18 respectively.

Therefore, we can conclude that the DCA can be used to discriminate between

Table 7.3: The effect of applying dynamic threshold on false positive values and true positive values for the MCAV.

Threshold	PhatE1		PhatE2.1		PhatE2.2		PhatE3		PmE1		PmE2		Total	
	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP
> 0.1	3	1	3	1	3	1	1	0	2	1	2	1	14	5
> 0.2	2	1	3	1	3	1	0	0	1	1	2	1	11	5
> 0.3	1	1	0	1	1	1	0	0	1	1	2	1	5	5
> 0.4	0	1	0	1	0	1	0	0	0	1	0	1	0	5
> 0.5	0	0	0	0	0	1	0	0	0	1	0	1	0	3

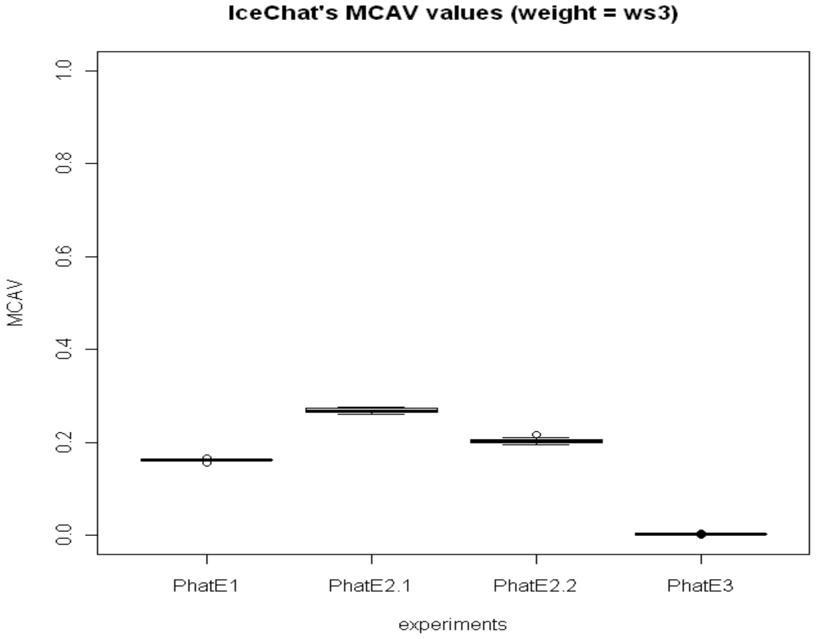

FIGURE 7.5: IceChat's MCAV generated by DCA using signal weight WS_3.

benign programs and P2P bots by generating higher MCAV/MAC values than those of the normal processes such as Firefox, Icechat and WASTE client. This implies that in addition to the detection of the bot itself the DCA can detect the performance of outbound scanning activity. Therefore the null hypothesis two (H2) can be rejected as in all cases the DCA successfully discriminates between normal processes and P2P bots.

TABLE 7.4: The effect of applying dynamic threshold on false positive values and true positive values for the MAC.

Thre-hold	PhatE1		PhatE2.1		PhatE2.2		PhatE3		PmE1		PmE2		Total	
	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	TP	FP
> 0.04	1	1	1	1	1	1	1	0	0	1	0	1	4	5
> 0.08	1	1	1	1	1	1	1	0	0	1	0	1	4	5
> 0.12	0	1	0	1	1	1	0	0	0	1	0	1	1	5
> 0.16	0	1	0	1	0	1	0	0	0	1	0	1	0	5
> 0.20	0	1	0	0	0	1	0	0	0	1	0	1	0	5

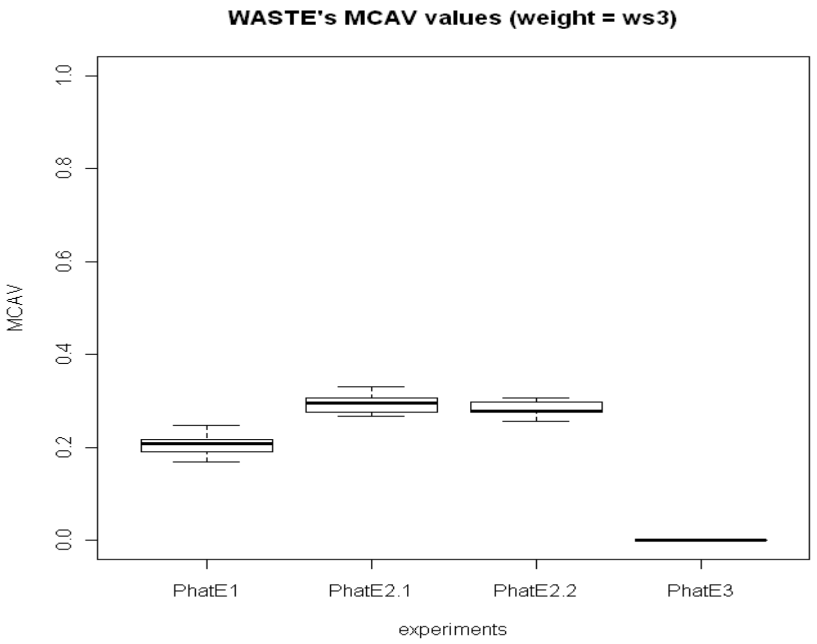

FIGURE 7.6: WASTE client's MAC values generated by DCA using signal weight WS_3.

Null Hypothesis Three (H3)

In addition, from Table 7.3 and Table 7.4 we can see that in all experiments, except for PhatE3, the number of false positive alarms generated when using the MAC value is less than the number of false positive alarms generated when using the MCAV value. From this we can conclude the MAC value reduces the number of false alarms in comparison to the MCAV value and the null hypothesis three (H3) is rejected.

7.4 Evaluation

In this section we evaluate our proposed bots detection method using DCA with other existing bots detection techniques. Because most of the currently existing techniques focus on botnets detection rather than bots detection and few bots detection techniques are available, the evaluation will be limited to the existing bots detection techniques.

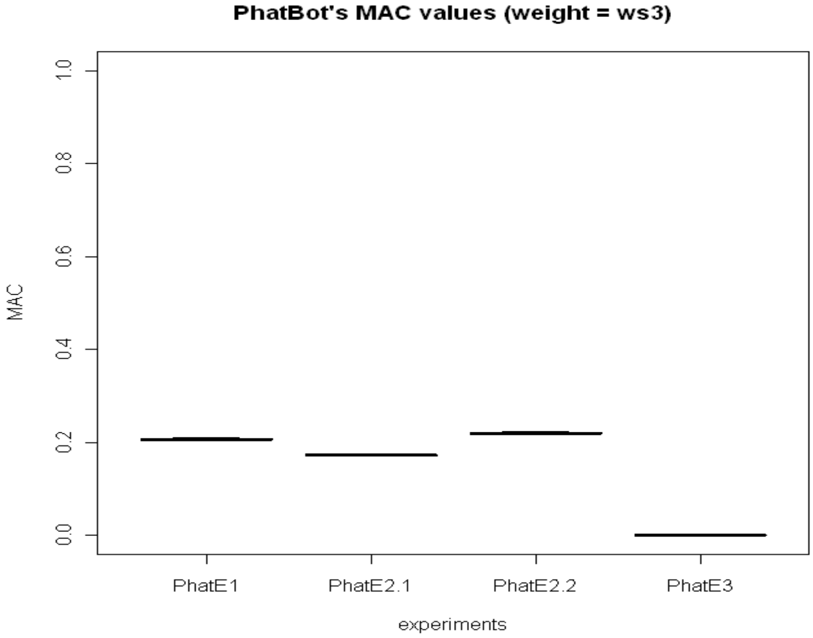

Figure 7.7: Phatbot's MAC values generated by DCA using signal weight WS_3.

One of the currently used technique to detect an individual bot running on a system is called BotSwat implemented by Stinson [151]. BotSwat is already explained in section 2.5.4 and the comparison between BotSwat and our framework is presented in section 2.6. We have requested the source code and the binaries from the author. Unfortunately, the source code of BotSwat was not documented properly. There are some points to consider when using BotSwat. The version that we have received does not automatically monitor all applications, thus, we need to specify which process we want, as our target. The second point is that after specifying the target, we need to examine the log files which contain large amount of data in order to report a bot detection event in the case of detecting a bot-like behaviour as mentioned by the author. The automatic detection of a bot is not provided by the version we have received from the author, therefore, it was difficult to examine these log files. The author also pointed out that in order to run BotSwat, one need to disable McAfee's Buffer overflow protection due to general interposition approach. The interposition approach describes how the original function calls can be replaced from a set of pre-

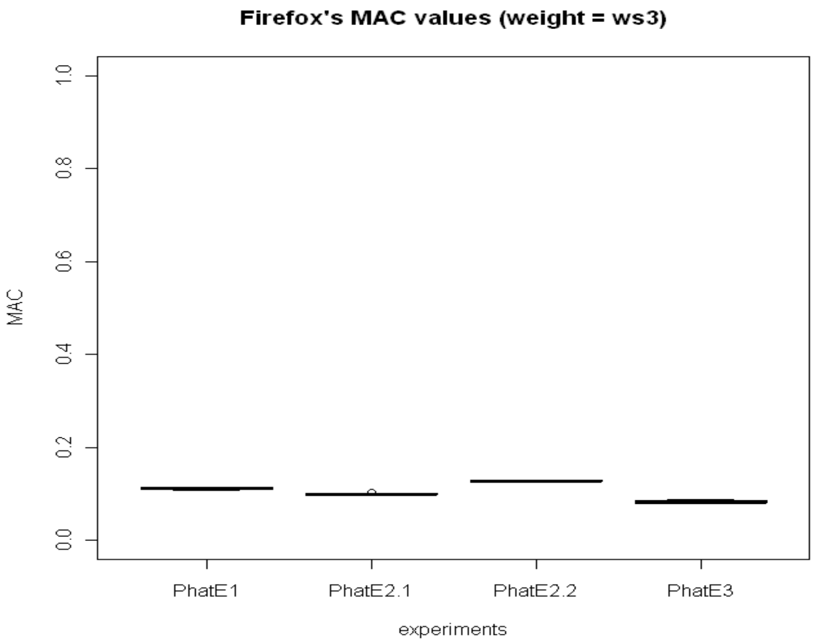

Figure 7.8: Phatbot's MAC values generated by DCA using signal weight WS_3.

defined function calls which are to be monitored [151]. For these reasons, it was difficult to compare our framework with BotSwat.

7.4.1 A non-DCA Algorithm

We have also implemented a non-DCA algorithm to compare the generated results with P2P bots results using the DCA. The same log files (signals and antigen) as for DCA are used for a non-DCA algorithm. In addition, the same experiments conducted are used in this comparison. These experiments are PhatE1 for idle scenario, PhatE2.1 for information gathering scenario, PhatE2.2 for attack scenario, PhatE3 for normal scenario, PmE1 for inactive Peacomm bot scenario and finally PmE2 for active Peacomm bot scenario.

The non-DCA method is based on two criteria. The first criteria is to analyse the antigen log file based on the frequency of API function calls generated by processes (i.e. calculate the number of function calls invoked per process). The second criteria is to analyse the signal log file by *setting a sensitivity value (SV)* for each signal

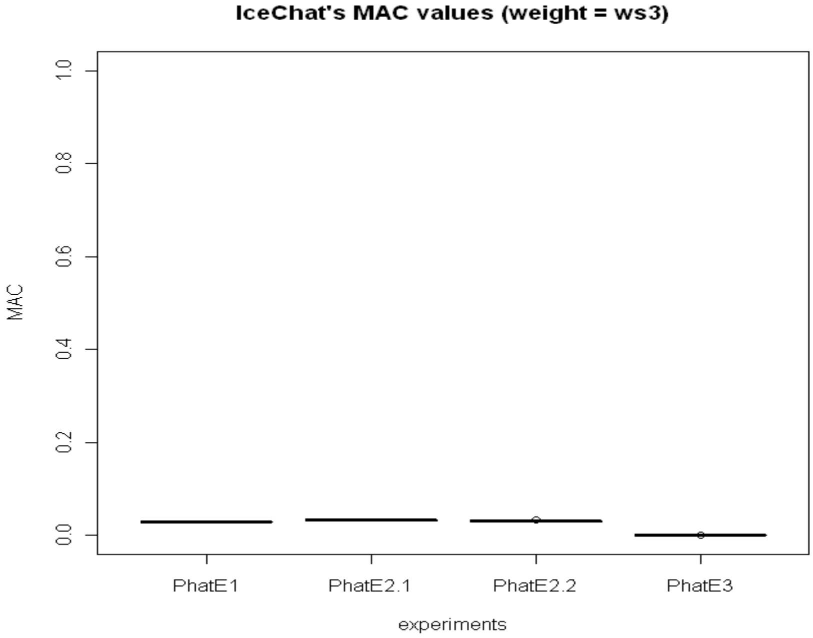

FIGURE 7.9: Icechat's MAC values generated by DCA using signal weight WS_3.

(PAMP, DS and SS).

The algorithm is described in Algorithm 4 works as follows. We set a sensitivity value (SV) and check if the values of PAMP, DS, and SS exceeds the specified SV. If the signal value exceeds the specified SV, we assign a value of *one* to its records, otherwise, we assign a value of *zero*. Then, we examine if signals' records have the same values, we assign a value of *one* which represents a correlation between the signals (PAMP, DS, SS) at that period of time. We repeat this process for all the signals in the signal log file.

Then, we calculate the anomaly factor and the correlation factor from the following equations:

$$AnomalyFactor(AF) = \sum_{i=1}^{n} \frac{(X_{PAMPi} + X_{DSi} + X_{SSi})}{3n} \tag{7.1}$$

$$CorrelationFactor(CorrF) = \sum_{i=1}^{n} \frac{Corr_i}{n} \tag{7.2}$$

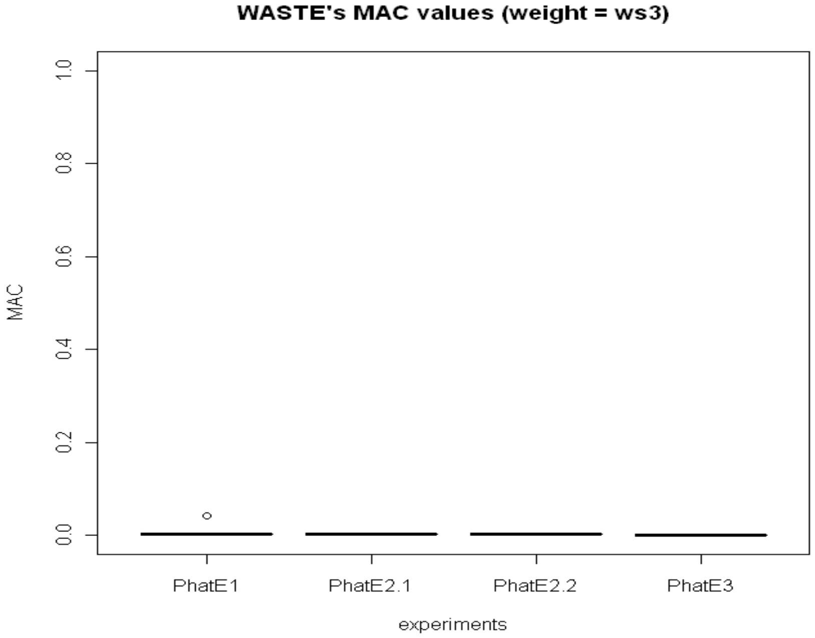

FIGURE 7.10: WASTE client's MAC values generated by DCA using signal weight WS_3.

where n is the time in seconds and X is the signal record which represents a logic value (zero or one) if the signal value exceeds a predefined sensitivity value (SV). The correlation factor represents how signals are related to each other and its range from zero to one. For example, if PAMP and DS have high values than sensitivity value (SV) and SS has a low value than (100-SV), (note that signal values are normalised from zero to 100, thus we change the SV from zero to 100.), this will generate a high correlation between these signals at that time. The final step is to calculate the *anomaly correlation value (ACV)* from the following equation:

$$ACV = AF * \exp(CorrF) \tag{7.3}$$

The use of exponential form to the correlation factor in this formula represents the confidentiality level of how signals are related to each other. For example, if the correlation factor is zero, this means that the signals in the log files are not correlated and the ACV will only depend on the anomaly factor (AF). If the correlation factor is

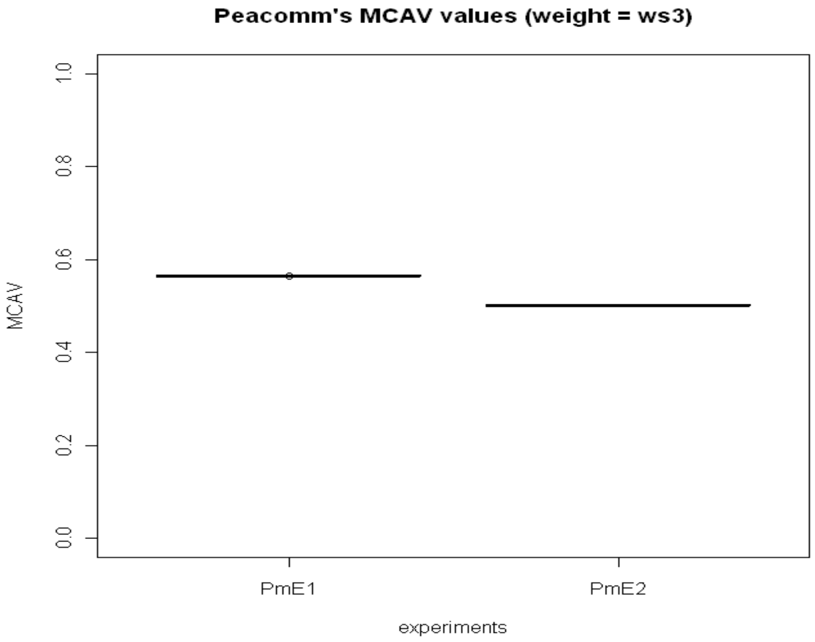

FIGURE 7.11: Peacomm's MCAV generated by DCA using signal weight WS_3.

higher than zero, the ACV will depend on both the anomaly factor and the correlation factor. Thus, the more correlation we have between signals, the higher the ACV will be. The maximum value of ACV is 2.7182 which is the value of exp(0) as the value of anomaly factor ranges from zero to one as well.

The results of applying this technique are shown in Table 7.5. In this table, the frequency of API function calls for each process for all the experiments conducted and the anomaly correlation values (ACV) when applying different SV are presented. As shown from this table, we note that changing SV value generates different anomaly correlation values (ACV). If we increase the sensitivity of the system by decreasing the SV, this will lead to the reduction of ACV as shown in Table 7.5.

To detect malicious activity, a threshold value is needed. For example, if we consider the case of $SV > 20$, setting a threshold value 20 detects all malicious activities on the system for experiments PhatE2.1, PhatE2.2, PmE1 and PmE2 except for PhatE1 which indicates a false negative case. For experiment PhatE3, the anomaly correlation value (ACV) is below 20% which indicates normal activity on a system as

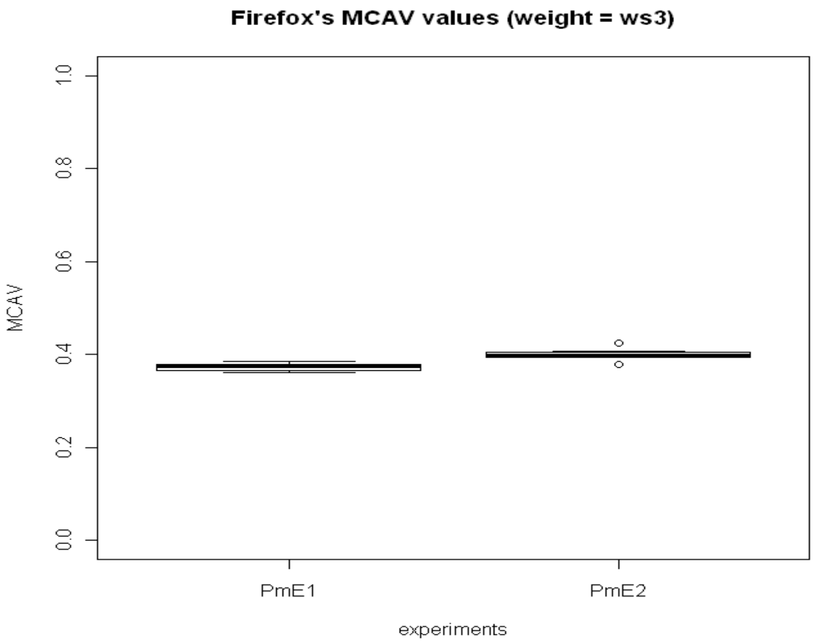

FIGURE 7.12: Firefox's MCAV generated by DCA using signal weight WS_3.

expected. Increasing the threshold to 25% and or 30% generates two false negative alarms (i.e. PhatE1 and PhatE2.1). Reducing the threshold value to 15 generates zero false alarms and 100% true positive alarms when setting SV to 20. In general, we have noticed that the best value of threshold for all SV (i.e. 10 to 50) to detect abnormal activities on the system is when using a threshold value of 10. Setting a threshold level to this value will generate zero false positive alarms and 100% true positive alarms for all experiments. This is shown in Table 7.6 and the ROC analysis is shown in Figure 7.19.

Another important question is to know which processes are malicious and which processes are normal. Using the frequency of API function calls for each process as an indicator, it was difficult to determine which process is normal and which process is malicious. For example, based on the frequency of API function calls, the Phatbot is a malicious process in experiments PhatE1, PmE1 and PmE2 but in experiments PhatE2.1 and PhatE2.2 the Firefox process is the malicious process.

Comparing the obtained results using a non-DCA algorithm with the DCA al-

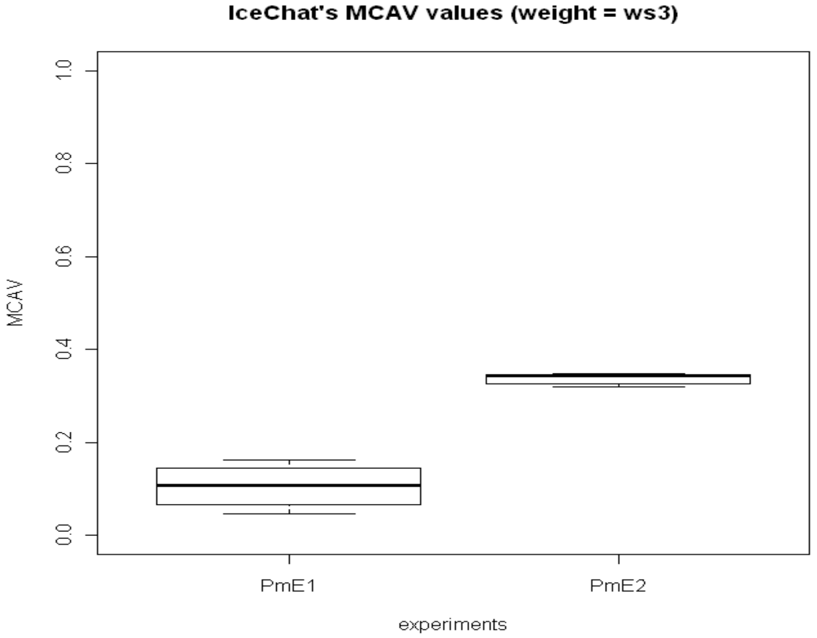

FIGURE 7.13: Firefox's MCAV generated by DCA using signal weight WS_3.

gorithm as in Chapter 7, we can see that the DCA algorithm has an advantage of classifying malicious processes on a system. This is indicated by generating higher MCAV/MAC values for malicious processes. In addition, the DCA produces a MCAV and MAC values for each process. In contrast, the non-DCA algorithm presents the system behaviour in general in which it can indicate abnormal activity on the system but it cannot classify malicious processes accurately. In addition, in the non-DCA algorithm the value of threshold to detect malicious processes is undefined as it is in the DCA and further experiments are needed to set a proper threshold for detecting malicious activity in the system.

7.4.2 Change of Bot's Behaviour Evaluation

In this section, we have conducted different experiments to show how resilient a DCA is to changes in bot's behaviour. We begin by examining the changes in the behaviour of Phatbot. We use the same dataset for the combined attack performed by the Phatbot where the Phatbot connects to the WASTE client and joins the channel,

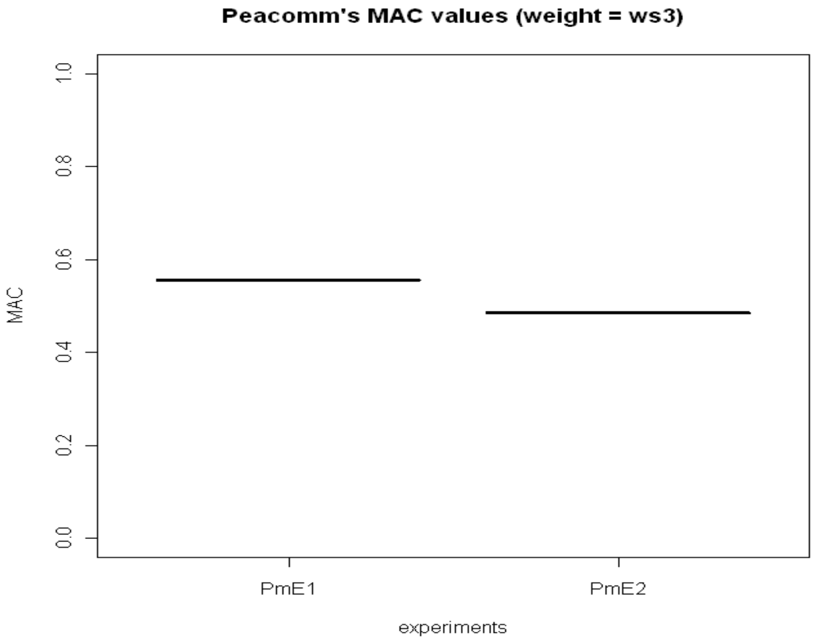

FIGURE 7.14: Peacomm's MAC values generated by DCA using signal weight WS_3.

the attacker starts to issue different commands such as SYN/UDP/ICMP flooding attack, obtaining sensitive information from a victim's machine, monitoring the user's activities, opening and deleting files as described previously in Section 7.3.2(experiment: PhatE2.2). These flooding methods are designed to emulate the behaviour of a machine partaking in a distributed denial of service attack. As part of the process of packet flooding the bots rely heavily on socket usage, as part of the packet sending mechanism. Note that during the flooding attack other 'normal' legitimate applications are still running.

We start by changing the PAMP signal and set all the values to zero which means that no PAMP signals are detected and we only have danger signals (DSs) and safe signals (SSs). We will call this experiment PhatE2.2.A. The second step is to set all danger signal values to zero. This is the case were no danger signal is being detected and only PAMP signal and safe signal are detected. We call this experiment PhatE2.2.B. The third step we have done is that we set all safe signals to the maximum value which is 100. In this case we remove the impact of safe signals on the other

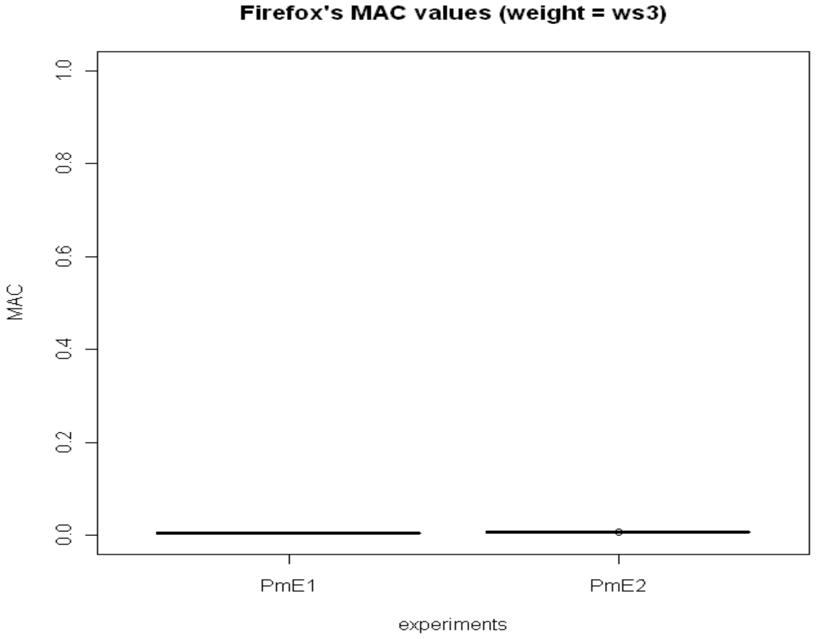

FIGURE 7.15: Firefox's MAC values generated by DCA using signal weight WS_3.

two signals, the PAMP and the danger signal. We call this experiment PhatE2.2.C. The last step we have performed is to swap the values of PAMP and danger signals and observe the effect of swapping these two signals on the detection performance. Each experiment is repeated ten times and the mean values are taken for the number of processed antigen, the MCAV and the MAC value. The results of changing the values of these signals is shown in Table 7.7.

From Table 7.7, we can see the effect of signal absence or changing the behaviour of bots on DCA detection performance. In experiment PhatE2.2.A the absence of PAMP signals did not have large impact on DCA detection performance. This is because the MCAV and the MAC value for bot is significantly higher than the other benign processes. The bot‘s MCAV value is higher than Firefox‘s MCAV value by approximately 61%. In experiment PhatE2.2.B, we have also noticed that the absence of danger signal did not have a large impact on DCA detection performance because bot‘s MCAV/MAC values are significantly higher than the MCAV/MAC values for other process. The MCAV of the bot is higher than the MCAV of Firefox by more than

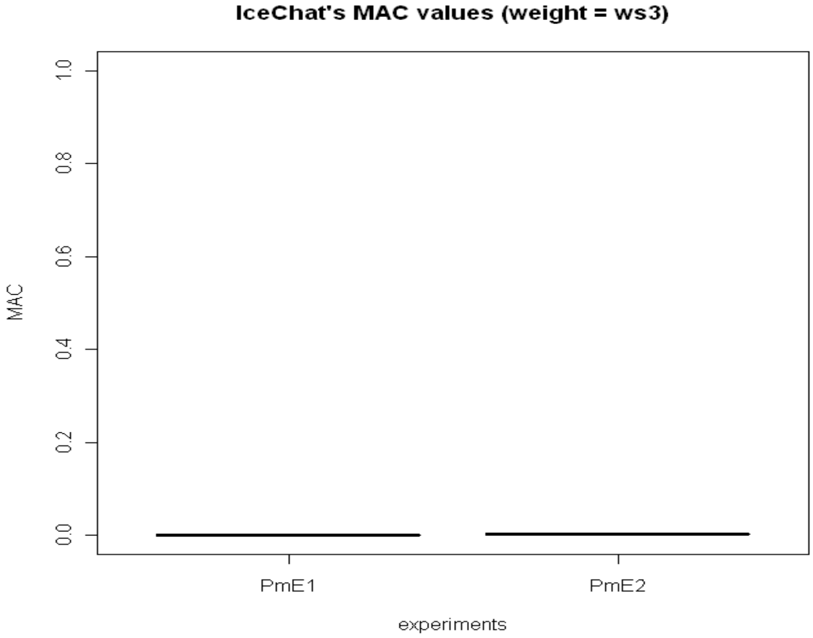

FIGURE 7.16: Firefox's MAC values generated by DCA using signal weight WS_3.

100%. In experiment PhatE2.2.C, where all the safe signals are set to the maximum values, the MCAV/MAC values for all the processes are zero. The obtained results suggest that the safe signal has large impact on DCs in comparison to other two signals which causes the DCs to be presented as semi-mature cells. As a result, the Phatbot is misclassified as a normal process. In the final experiment, PhatE2.2.D, we can see that although there is an increase of the MCAV/MAC values for all processes, the Phatbot's MCAV/MAC values are still significantly higher than the rest of the benign programs. Thus, the affect of swapping the PAMP and danger signal values increase the MCAV/MAC values but does not have negative impact on the results.

The next scenario is to examine the effect of changing P2P bot's behaviour on the DCA detection performance. We also use the same dataset for Peacomm bot as in Section 7.3.2, Experiment PmE2. In this experiment, the Peacomm bot is executed and at the same time the user uses Firefox for browsing and checking emails and Icechat for having conversation with other users. We change the values of PAMP, danger and safe signal in a similar way to the previous experiments. First we set

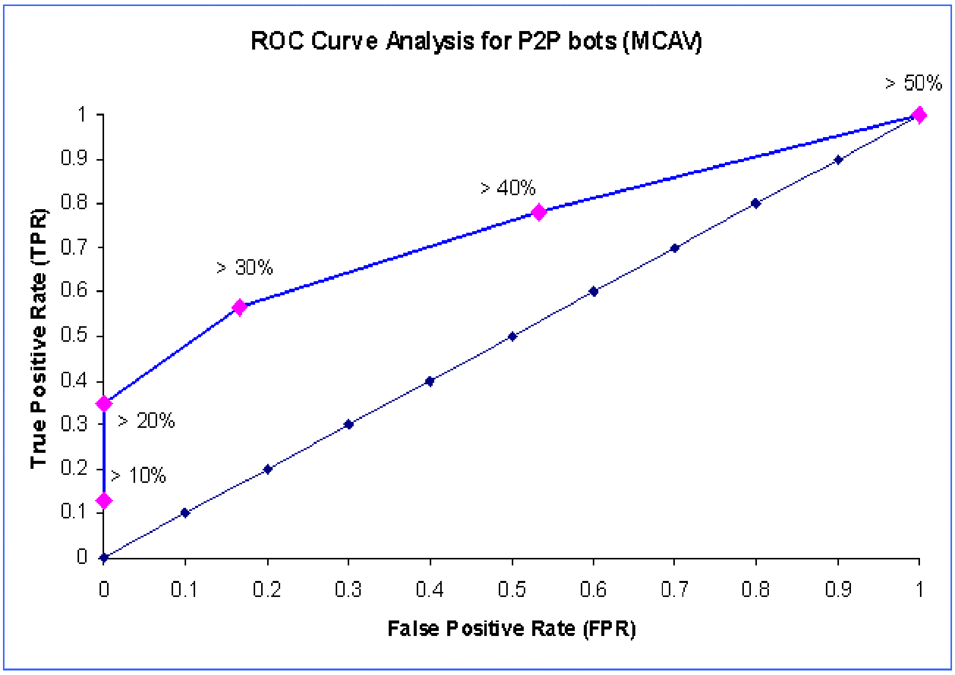

FIGURE 7.17: The affect of applying a dynamic threshold values on the MCAV for all experiments.

the value of PAMP signal values to zero and examine its effect on the DCA detection performance to represent the absence of PAMP signal. Then, we set the danger signal values to zero and examine its effect on detection performance which represents the absence of danger signal. Third, we set the safe signal values to the maximum (100) and examine its effect on the DCA detection performance. Finally, we also perform swapping between the PAMP and danger signals to analyse the effect of swapping these two signals on the DCA detection performance (experiment PmE2.D). Each experiment is repeated ten times. The results of these experiments are shown in Table 7.8.

From Table 7.8, we notice that setting the PAMP signal values to zero in experiment Pm2.A did not have large impact on the DCA detection performance and we can conclude that using the danger signal and the safe signal is sufficient to detect malicious activity on the host. This is because the MCAV/MAC values of the Peacomm bot are higher than the MCAV/MAC values for other benign processes. In

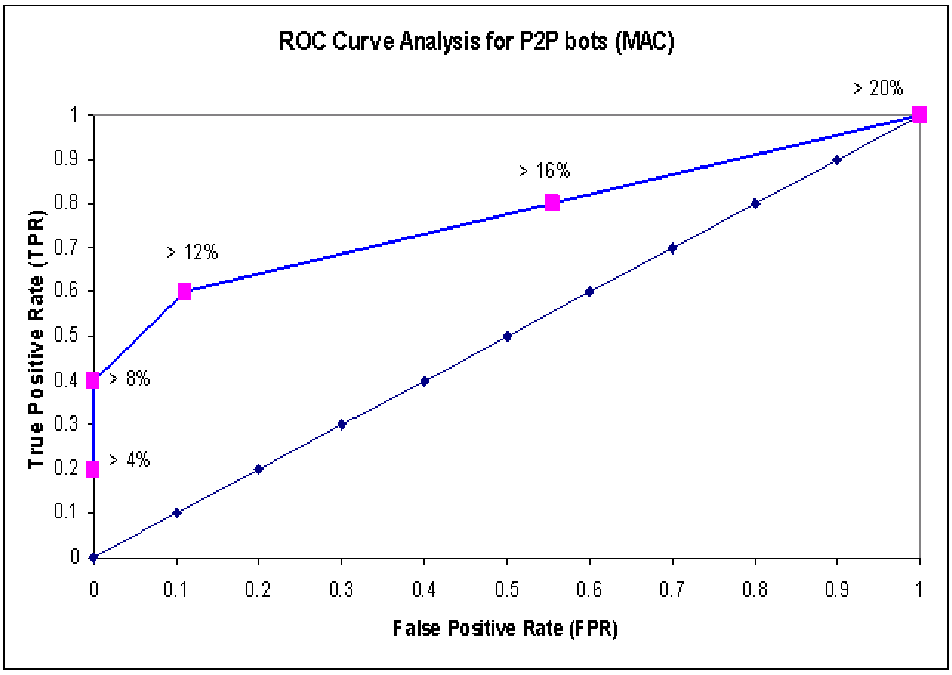

FIGURE 7.18: The affect of applying a dynamic threshold values on the MAC for all experiments.

experiment Pm2.B where all the danger signals are set to zero, we found that PAMP signal alone did not generate high MCAV/MAC values for the Peacomm bot in the absence of danger signal. We also noticed that the MAC value is significantly higher than the MAC values for other benign processes in comparison to the MCAV. In the third experiment, Pm2.C, we noticed a similar situation as in PmE2.B. Both the MCAV and the MAC values generate low values which indicate normal activities on the system and thus leading to a misclassification of bots as a malicious process. The final experiment, PmE2.D, although these is a large increase of MCAV/MAC values for all processes, the Peacomm bot has higher MCAV/MAC values in comparison to other benign programs. The difference is clearly obvious when applying the MAC value to compare the malicious process with the other processes. From the conducted experiments, we conclude that the three signals are complementary to each other to enhance the detection performance. The absence of one signal may affect the performance of the DCA detection and may leads to misclassification of processes. We also

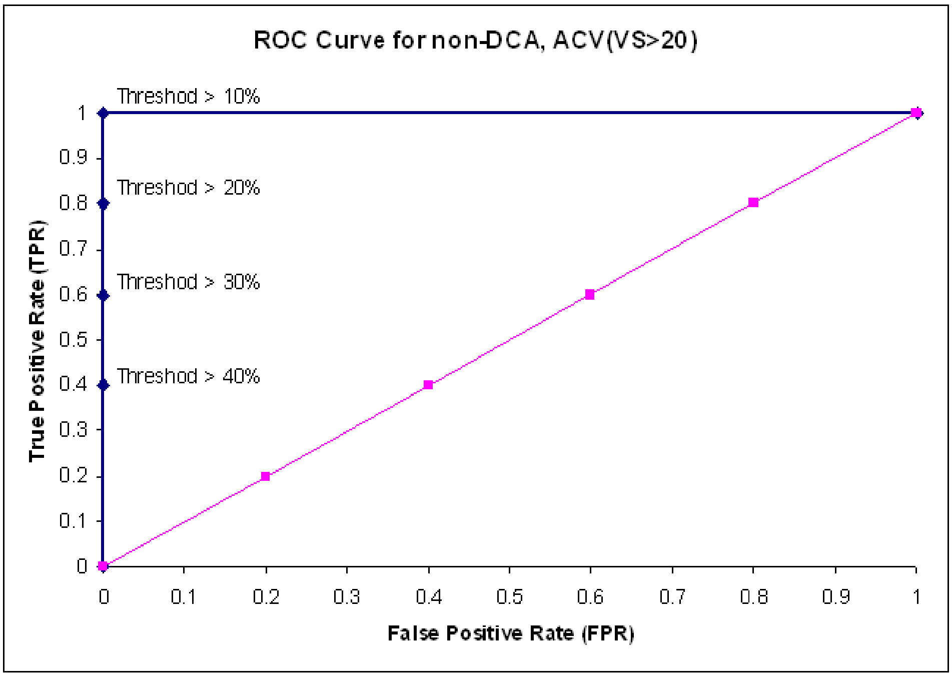

FIGURE 7.19: The ROC analysis for applying a dynamic threshold values on the anomaly correlation value (ACV) for VS=20.

conclude that the MAC value has a better abnormality comparison between processes.

7.4.3 DCA for Detecting other Malicious Software

The DCA has been used in various security areas such as detecting port scan [52] and SYN scan [49].In this section, we want to examine if the DCA will be able to detect other malware such as viruses and worms. To perform the experiment, we set up a honeypot at home with a clean host connected to the Internet and allow other malware to infect our host. The conducted experiment is performed on Windows XP machine without any Service Pack and it has 1.0 GHz processor. The APITrace was running on our host to monitor the activities of the malware similar to P2P bots detection experiments. The parameters used in DCA for this experiment are the same for P2P bots detection. The PAMP signal and the danger signal are also the same as in P2P bots detection with the exception of safe signal to represent the inverse rate of change of the amount of traffic sent per second. The PAMP signal represents the combined value of the number of destination unreachable (DU), number of connection reset (RST) and the number of failed connection attempt (FCA). The danger signal represents the rate of change of the amount of traffic sent every second. The normalisation of these signals is performed in a similar way to P2P bots detection experiments. Note that no user activity is performed in this experiment which was run for the duration of an hour.

Results

Once the host is connected to the Internet, the host is infected with a virus or a worm and it starts to send large amount of traffic over the network. Using our monitoring program APITrace, we have logged the three signals PAMP, danger and safe signals with the antigen executed by the worm/virus as shown in Figure 7.20 which represents the PAMP signal and Figure 7.21 which represents the danger signal. In both figures, the x-axis represents the time in seconds. We can see from Figure 7.20 that there is an increase in the number of DU, RST and FCA. In this situation, the virus/worm tries to make many connections to different places but it seems that many connections failed because either they are behind a firewall or they do not exist any more. For the danger signal (DS) in Figure 7.21, we notice there is a sudden increase in the amount of traffic sent after the infection and this traffic lasts till the end of the experiment.

After signals are normalised as mentioned previously, these signals are combined and sorted in a timely manner with the collected antigen and passed to the DCA to obtain the results. The result is shown in Table 7.9.

From Table 7.9, we can see that the process with ID number 1676 (the virus/worm) has high values for MCAV and MAC, 0.8249 and 0.6662 respectively, in comparison to the MCAV/MAC value of other processes. The results of this experiment show that the DCA can detect malicious software other than IRC/P2P bots. We conclude that the DCA can be applied to detect different kinds of malicious softwares but the selection of signals is important to gain a high detection performance and effective results.

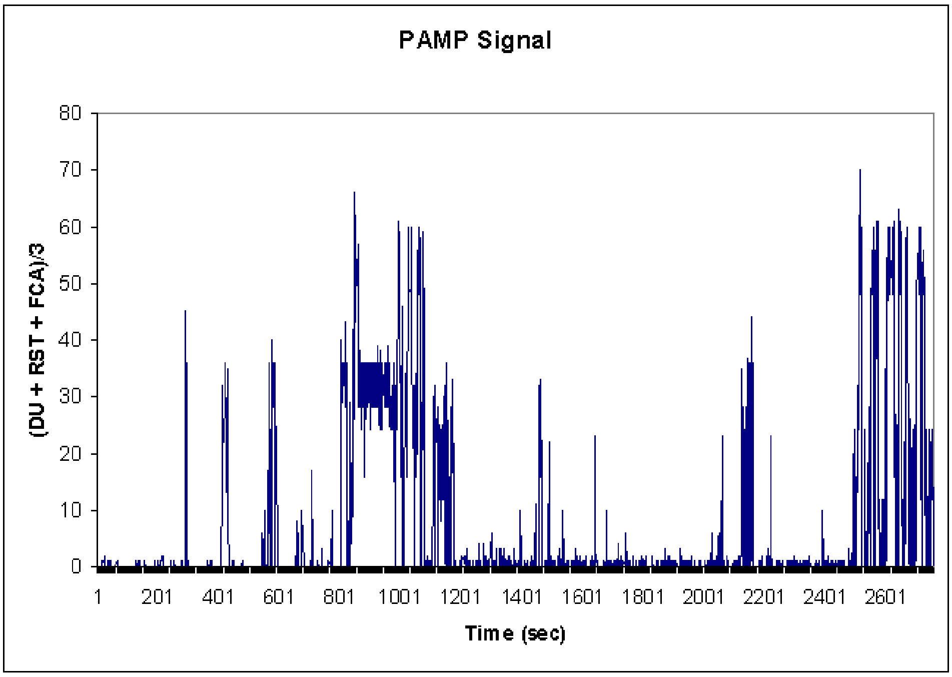

FIGURE 7.20: The PAMP signal used for DCA input to detect other malicious software.

7.5 Summary and Conclusions

In comparison to IRC bots, Peer-to-Peer (P2P) bots are more difficult to monitor, detect or shut down as there is no central command and control structure and most of the traffic is encrypted. One way to detect such bots is by monitoring and correlating different activities on a machine. In this chapter, we use the DCA as an intelligent correlation algorithm to correlate different behaviours of normal processes and P2P bots. This correlation of behaviours is based on specifying signals combined with antigen. The choice of proper signals enhances the distinction of normal from malicious processes. Two case studies are used in this chapter to measure the performance of DCA, Phatbot and Peacomm bot. In both cases, the results show that the DCA is able to classify bots as abnormal processes in comparison to benign processes by generating significant differences in the MCAV/MAC values for both normal and abnormal processes.

We also implemented a non-DCA algorithm to evaluate the performance of the

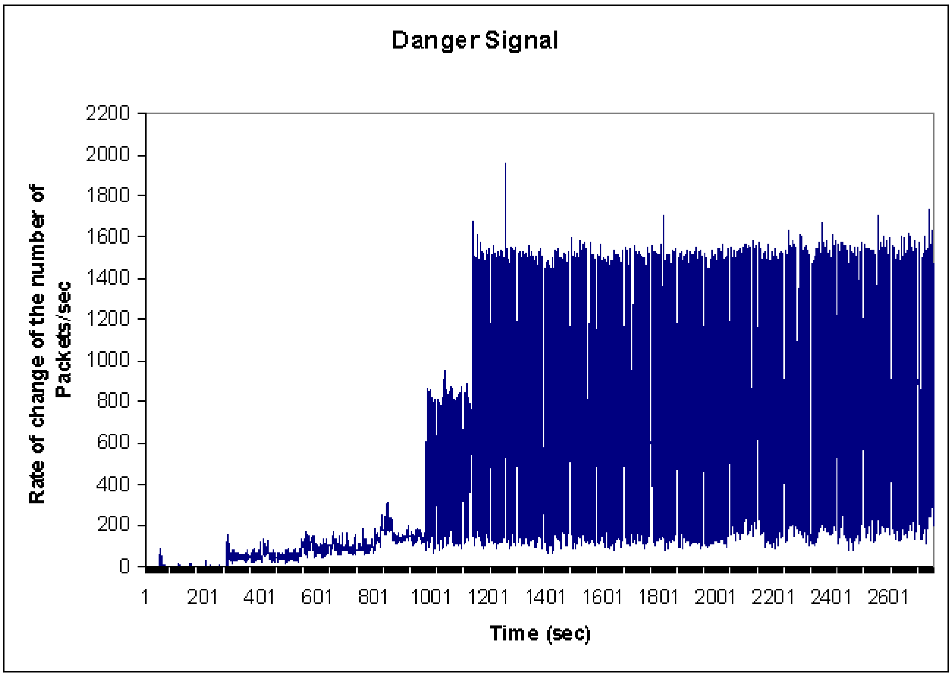

FIGURE 7.21: The danger signal (DS) used for DCA input to detect other malicious software.

DCA. Although the results from the non-DCA algorithm show the detection of the abnormality on the system with a very low false positive rate, the DCA has an advantage of classifying processes into normal and malicious. This specialty is not provided by the non-DCA algorithm as it presents the situation in general. In addition, we perform different experiments to show how resilient the DCA is to the changes in bots and can the DCA detect malware other than bots. Our results show that the DCA can still detect bots in most cases even if there are changes in bot‘s behaviours. This mainly depends on the proper selection and categorisation of signals as input to the algorithm. In the last experiment, we also show that the DCA is capable of detecting malware other than bots. As a result, the DCA can be applied to different security areas.

```
input  : S= (PAMP, DS, SS)
Initialise SV;
for $i$ = 1 to $n$ do
    if $PAMP_i > SV$ then
        $X_{PAMPi} = 1$;
    else
        $X_{PAMPi} = 0$;
    end
    if $DS_i > SV$ then
        $X_{DSi} = 1$;
    else
        $X_{DSi} = 0$;
    end
    if $SS_i > SV$ then
        $X_{SSi} = 1$;
    else
        $X_{SSi} = 0$;
    end
    if $X_{PAMPi} = 1$ and $X_{DSi} = 1$ and $X_{SSi} = 1$ then
        $Corr_i = 1$;
    end
end
```

Algorithm 4: A non-DCA algorithm

TABLE 7.5: The results of using a non-DCA algorithm when (1) applying different sensitivity values (SV) to calculate the Anomaly Correlation Value (ACV) and (2) considering the frequency of API function calls per process for P2P bots.

Experiment	Process	Frequency	ACV (>SV)				
			SV=10	SV=20	SV=30	SV=40	SV=50
PhatE1	Phatbot Firefox Icechat WASTE	4851 4438 2101 191	0.2504	0.1858	0.1408	0.1137	0.1032
PhatE2.1	Phatbot Firefox Icechat WASTE	6315 11844 2063 172	0.3171	0.2367	0.1757	0.1422	0.1296
PhatE2.2	Phatbot Firefox Icechat WASTE	7392 9068 2758 199	0.4278	0.3279	0.2323	0.1619	0.1460
PhatE3	Phatbot Firefox Icechat WASTE	0 5754 1790 0	0.0541	0.0500	0.0358	0.0265	0.0207
PmE1	Peacomm Firefox Icechat	617349 9902 71	0.6856	0.6053	0.5226	0.4503	0.3404
PmE2	Peacomm Firefox Icechat	628838 4449 9464	0.6930	0.6374	0.5638	0.4456	0.3366

TABLE 7.6: The effect of changing threshold value on false positive and true positive values for ACV.

Threshold	VS=10		VS=20		VS=30		VS=40		VS=50	
	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP
> 10	0	5	0	5	0	5	0	5	0	5
> 20	0	5	0	4	0	3	0	2	0	2
> 30	0	3	0	3	0	2	0	2	0	2
> 40	0	2	0	2	0	2	0	2	0	0
> 50	0	2	0	2	0	2	0	0	0	0

Table 7.7: The effect of changing signal values for experiment PhatE2.2 on the MCAV/MAC values.

Experiment	Process	Processed Antigen	MCAV	MAC
PhatE2.2.A PAMPs = 0	Phatbot	6636.2	0.5079	0.2054
	Firefox	6346.5	0.3145	0.1216
	IceChat	2486.1	0.1979	0.0299
	WASTE	180.8	0.2722	0.0029
PhatE2.2.B DSs = 0	Phatbot	6628.7	0.4185	0.1679
	Firefox	6450.0	0.1902	0.0742
	IceChat	2508.1	0.1547	0.0235
	WASTE	174.2	0.1954	0.0200
PhatE2.2.C SSs = 100	Phatbot	6509.1	0.0000	0.0000
	Firefox	6250.3	0.0000	0.0000
	IceChat	2453.9	0.0000	0.0000
	WASTE	176.3	0.0000	0.0000
PhatE2.2.D $PAMP \leftrightarrows DS$	Phatbot	6554.3	0.6397	0.2577
	Firefox	6289.7	0.4609	0.1781
	IceChat	2462.0	0.2607	0.0394
	WASTE	183.3	0.3989	0.0049

Table 7.8: The effect of changing signal values for experiment PmE2 on the MCAV/MAC values.

Experiment	Process	Processed Antigen	MCAV	MAC
PmE2.A PAMPs = 0	Phatbot	136957.5	0.4740	0.4585
	Firefox	2711.8	0.3862	0.0074
	IceChat	1468.4	0.3011	0.0031
PmE2.B DSs = 0	Phatbot	154840.2	0.0842	0.0816
	Firefox	2976.0	0.0645	0.0012
	IceChat	1564.4	0.0939	0.0009
PmE2.C SSs = 100	Phatbot	126993.2	0.0205	0.0198
	Firefox	2552.9	0.0168	0.0003
	IceChat	1321.1	0.0181	0.0001
PmE2.D $PAMP \leftrightarrows DS$	Phatbot	131106.6	0.9334	0.9029
	Firefox	2547.5	0.6564	0.0123
	IceChat	1416.2	0.6659	0.0069

TABLE 7.9: The mean MCAV/MAC values generated from DCA using signal weight WS_3 for a virus detection

Experiment	Process	Processed Antigen	MCAV	MAC
Honeypot	1468	468.4	0.0000	0.0000
	344	36.2	0.0000	0.0000
	616	936.5	0.3658	0.0316
	1676	11926.9	0.8249	0.6662
	1808	1401.2	0.3840	0.0364

Chapter 8

Summary, Conclusion and Future Work

8.1 Conclusions

The Internet is persistently threatened by the increasing use of different types of malicious softwares (malware) such as viruses, worms and recently bots. These malware have a negative impact on both the Internet network and the personal computers. The effect of attacking the Internet results in network delays due to congestion, extensive waste of network bandwidth which costs companies billions of dollars. On the other hand, the impact of attacking personal computers can include corrupting users' computers and data and stealing sensitive information. In the past few years we have seen an exponential growth in the usage of bots for malicious activities which poses severe problems for both the Internet and the users. Our motivation is to investigate this area and find a suitable solution for this kind of threat.

In this work we presented a framework as a host-based bonets/bots detection system. The detection of botnets or bots on the infected machine is performed by correlating bots' behavioural attributes. The concept of correlating attributes within a specified time-window increases the level of detection of malicious behavioural activities as depending on one process attribute may generate large number of false alarms. This also leads to the challenge of choosing the right correlation algorithm which enhances the detection of malicious programs. Our framework is applied to various bots command and control structures such as IRC bots and P2P bots. An evaluation of the framework showed success in detecting malicious behaviours on the system.

8.2 Evaluation of Aims

In section 1.2, we have presented our hypotheses which lists questions that we want to address in this work. These hypotheses are summarised below:

1. *Do we use intrusion or extrusion detection system?.*

2. *What type of detection method do we use and how we are going to collect our data?.*

3. *Are we focusing on one activity or different activities?.*

4. *Is our framework suitable for botnet Detection?.*

5. *Can our framework detect individual bots and is detecting an individual bot more important than detecting botnet?.*

6. *Can our framework detect different types of command and control structures other than IRC bots such as P2P bots?.*

7. *How to evaluate our framework?.*

For the first point of our hypotheses, because our main focus in this work was to detect bots running on a compromised host, we thought that using the concept of extrusion detection will be more suitable for us to achieve this task. This is due to the fact that we are trying to recognize malicious activity by inspecting and correlating user activities and process activities with outbound traffic. Thus, we are not trying to prevent the attack using well know techniques such as filtering inbound traffic and using well known bot‘s signatures to analyse network traffic and we did not use pre-defined signatures to detect the running bots. Our assumption is that the bots are already installed on the victim's hosts through an accidental opening a malicious attachments in the email, which contains a bot or by visiting malicious websites and downloading the bot. As a result, the concept of extrusion detection system was more suitable for botnet/bots detection.

For the second hypothesis, although signature-based detection techniques, which are used by most anti-virus softwares or by existing botnet detection techniques, are very useful to detect existing bots we believe that they will fail in case if new bots with unknown signatures appear. In addition, it will take time to update the databases of all other anti-virus softwares with the new signatures. As a result, it was important for us to detect these bots as fast as possible before they start their activities with low false alarms. For these reasons, we have decided to use behavioural detection techniques as a solution for detecting malicious activities by correlating similar activities from multiple sources to detect botnet and correlating different activities from local host to detect the malicious bot. From our results in detecting botnet, IRC bots and P2P bots, we believe that correlating bot's behaviour within a specified time period can increase the detection performance. We have collected our data by monitoring and intercepting specified API function calls executed by bots. These function calls represent the activities of a process used to achieve some tasks. We found that monitoring these function calls can lead us to monitor the behaviour of processes.

In the third hypothesis and based on observing the behaviour of bots, we noticed that bots can perform multiple malicious activities within a short time period such as keylogging and issuing denial of service attack (discussed in Section 3.6). We have mainly concentrated on three activities: (1) User activity, (2) Process activity and (3) Network communications activity. In terms of user activities, these include intercepting API function calls which relates to user keystrokes, mouse movements and screen capturing. For process activities, we monitor API function calls which relate to file access and registry access. The final activity includes monitoring network information events such as time differences between data received or sent over the network, time difference between sending information to the outside world, which processes use network communications, number of destination unreachable messages, number of failed connection attempts messages and number of connection resets messages. Note that we are not interested in the payload of packets as the packets can be encrypted. We noticed that one activity may not be enough to provide evidence to detect the existence of bots, therefore, monitoring different activities can enhance our detection performance. We also noticed that if the monitored activities have some sort of causality such as intercepting user keystrokes, storing the intercepted keystrokes and sending them to the attacker within short time period produce high anomaly factor than if these activities are disordered or executed in different time periods. As a result, we found that the relationship and the correlation between these activities will increase the detection performance and reduce false alarms.

To address the forth hypothesis, we started our work by developing a framework to detect botnet activities to see how difficult it is to detect such phenomena (Chapter 4). In comparison with most of the current research work done which use a signature-based approach to detect botnets, our framework does not use any predefined signatures to detect botnet, but rather it is based on the reaction behaviour of the bots. The algorithm simply works by correlating log files from different resources to detect botnet activities. The correlation is based on monitoring similar activities generated by bots when responding to the botmaster commands. These activities are generated when bots invoke one or many function calls to perform specific task simultaneously. These function calls with their arguments are stored in log files. The log files are monitored for any changes that have happened during bots actions and the correlation between these changes is used to report botnet activities. From the obtained results, we found there is a high correlation between bots activities which leads to a successful detection of botnets. Although the correlation algorithm does not require any pre-defined signatures and can be applied to detect different types of bots using different command and control structures, it has many issues which can effect the detection performance. First, the algorithm can be defeated if the attacker designs his bots to respond in different time periods which makes that correlation process difficult. In addition, the number of monitored hosts can influence the correlation algorithm. Monitoring large number of hosts can increase the probability of finding hosts participating in malicious activities such as a DDoS attack. A more intelligent correlation algorithm which takes into account the amount of changes in log files is needed to enhance the detection.

In the fifth point of our hypotheses, we hypothesised that detecting an individual bot is more interesting and might be more challenging than detecting botnets. In our research, we show that most of the current research is aimed at botnet detection. Little research has been done in detecting an individual bot on a host. One of the reason is that botnets generate large traffic used to perform denial of service attack and similar activities can be noticed on the network such as joining the same IRC channel, connecting to the same port or using names with similar prefix. Thus, detecting this kind of behaviour is not difficult. A more difficult task is to detect an individual bot running on a host. As we have already shown, there are many reasons for this claim. In addition to participating in denial of service attack, the bot can be used for other malicious services. One of these malicious services includes intercepting the user sensitive information by means of keylogging activity. The user might not be interested if his machine participates in denial of service attack but for sure s/he will have lots of worries if his personal information, credit card numbers or personal bank details are used by other parties.

For the purpose of detecting an individual IRC bot running on a host (Chapter 5) and based on our observation of malicious bots behaviours, we noticed that many bots are implemented with malicious activities such as keylogging. The keylogging activity is an interesting activity for the attacker and represents a serious threat to the users because the bot can intercept all user‘s keystrokes as well as taking a screen capture of the system every second which means that every move by the user is recorded. The bot can store this information and send it to the attacker. This was the main reason to detect an individual bot running on a system by correlating these activities. To collect our data, we have developed a monitoring program which intercepts specified function calls and saves these function calls with other parameters in log files. The method used in this work is to monitor function calls executed by processes which represent their activities and then correlate these activities within a specified time period. Based on time, we use a simple correlation method called Spearman’s rank correlation (SRC) algorithm to perform this task. Although the results were promising and met our aim of detecting an individual bot on a system, we faced the problem of having large number of false positive alarms due to idle periods generated when no activities are noticed. In addition, our results showed that detecting bots by monitoring the keylogging activity alone could generate false alarms as some of the benign programs may use the same function calls. As a result, more than one activity is needed to reduce the false alarms which support the third point in our hypotheses for the need for multiple activities to detect bots. Moreover, we also noticed that these activities should be related to each other in a causal manner in order to improve the detection performance. For example, if the bot cannot communicate with the attacker, the keylogging activities do not represent a threat to the user even though the bot is still intercepting the user's keystrokes.

Because of the large number of false positive alarms generated by idle periods when using SRC algorithm, we needed a more intelligent way of correlating different activities of processes on a system. This leads us to use the Dendritic Cell Algorithm (DCA) (Chapter 6). DCA has been applied to many problems particularly in the area of intrusion detection in computer security. The DCA is a more intelligent way of fusing and correlating information from disparate sources. The immune inspired DCA is based on an abstract model of the behaviour of dendritic cells. These cells are the natural intrusion detection agents of the human body, which activate the immune system in response to the detection of damage to host tissues. As an algorithm, the DCA performs multi-sensor data fusion on a set of input 'signals', and this information is correlated with potentially anomalous 'suspect entities' which we term 'antigen'. This results in a pairing between signal evidence and suspects, leading to information which will state not only if an anomaly is detected, but in addition the culprit responsible for the anomaly.

To collect our data, we used our monitoring program to intercept API function calls. The choice of selecting our input data is crucial on the performance of our correlation algorithms. We focus on three different activities to collect our input data which are the same activities used to detect bots by applying SRC algorithm. These three activities include user activities, process activities and network information in real time. For example, the PAMP signal defines the keylogging activity and file access. The danger and safe signals define how fast the bot responds to the attack commands and how fast it invokes the same function calls respectively. The danger and the safe signals depend on intercepting communication function calls. These function calls are saved with other parameters in log files. Finally, the data in log files are pre-normalised and passed to the correlation algorithms for processing and analysis.

From the obtained results, we noticed that the DCA achieved a better detection performance in comparison to the SRC algorithm in detecting an individual IRC bot on a system with reduced number of false positive alarms as shown in Chapter 6. In addition, the DCA has an advantage of stating not only if an anomaly is detected, but also states the culprit responsible for the anomaly in comparison to the SRC algorithm. The original DCA is based on producing the Mature Context Anomaly Value (MCAV) to categorize processes into either normal or abnormal. In this work, we introduced the modified MCAV, termed MAC value, which takes into account the frequency of API function calls executed by a process in proportion to all other active processes. This ensures that the active process has a higher weight than inactive processes. In comparison to MCAV, the results show that the MAC value reduces the number of false alarms which offers a better way of classifying normal and abnormal processes. We expect that the MAC performance will always be better than the MCAV except in certain cases. For example, the MCAV value for a monitored process will produce a high value indicating a real malicious activity but the number of antigen being processed is low due to miss proper selection of antigen. In this case, the MAC value will be low in comparison to the MCAV which may be considered as false alarm. We also examine the effect of changing DCA parameters, especially signal weights, on the performance of DCA to classify normal and abnormal processes. We concluded that choosing the right weights has a significant difference on the classification of malicious processes.

For the sixth hypothesis, we extended our research of bots detection to monitor the behaviour of P2P bots running on a system using the DCA (Chapter 7). Two case studies were used to perform this task, namely Phatbot and Peacomm bot. We collected our data following the same procedure as mentioned in Chapter 6 for both signals and antigen with modifications of PAMP and danger signals. The selection of these signals was based on observing the behaviour of P2P bots. For example, the P2P bots generate large number of destination unreachable messages (DU) or connection reset messages (RST) when trying to communicate with other peers. These signals are choosen to be our PAMP signal. In addition, P2P bots generate large amount of traffic for the same reason which we considered it as our danger signal. Our results showed that the DCA not only detects IRC bots but also other Command and Control structure bots such as P2P bots with low false positive alarms and high true positive alarms.

In the final point of our hypotheses, we compared the results obtained from the DCA when passing P2P data with the results generated from a non-DCA algorithm to evaluate the DCA detection performance (Section 7.4). It was noted that although the non-DCA algorithm was able to detect malicious behaviour in a system, it was difficult to classify processes which cause the abnormality. In addition, further experiments are needed to set a proper threshold value to detect the abnormality in the system because the threshold value was set after obtaining the results. In contrast, the DCA has a pre-defined threshold to detect the abnormality and can classify the processes into normal and abnormal with a high percentage of accuracy.

In Section 7.4, we also performed different experiments to show how the DCA is resilient to the changes in bots‘s behaviour. We started by examining the effect of a signal absence on the DCA detection performance. Our results showed that the safe signal is needed all the time because it suppresses the effect of the other two signals. The DCA can still detect malicious activity on our system in case of the absence of PAMP signal or danger signal in most cases. We also examined the effect of swapping the PAMP and the danger signals on the DCA detection performance. We found that swapping these two signals can influence the detection performance by increasing the MCAV/MAC values for all the processes. In such a case, the MAC value can produce a clear distinction of anomalous process. In addition, we examined the ability of the DCA to detect malware other than bots. Our results show that the DCA can detect other malicious softwares and can be applied to different security areas.

In summary, the use of DCA as an intelligent correlation algorithm offers several advantages in the area of bots detection. The first advantage is that the DCA performs both data fusion of processing signals and correlates these signals with suspect antigen. Secondly, no pre-defined signatures are needed and there is no training phase in order to detect the existence of a bot. Because of this, a minimal amount of data processing and resource consumption is required in comparison to signature-based bots detection algorithms. Finally, in terms of complexity and portability the DCA has not only a low complexity but achieves high performance of anomaly detection. One disadvantage of the DCA is that appears when low number of antigen is monitored for a process it may lead to miss classification. We showed that this problem can be solved by using the MAC value instead of the MCAV to present our results. In addition, in case of having two highly active processes, the DCA will not be able to distinguish between these processes and thus generate false alarms.

8.3 Critical Assessment of our Work

Although there has been much research in the field of detecting botnets, few have focused on detecting an individual bot running on a system. Our main focus is to provide a framework that detects an individual bot as well as botnet. In our research, we did not attempt to prevent the malicious bots from breaking into our system using signature-based techniques, but rather detecting the attack by observing malicious behaviours on a system by correlating different activities.

The results we obtained were promising in detecting malicious activities on a system but there are few points to be considered for further improving bot detection:

- Our monitoring program which collects the data is running on the same compromised host. If the monitoring program is discovered by any means, the attacker will try to prevent the monitoring program from collecting the data by compromising it or shutting it down.

- The monitoring program is running on the user-land area. The attacker can control this area once the host is infected. A better way of implementing the monitoring program is by implementing and running the monitoring program in a kernel-land area. If the attacker tries to stop the monitoring program, it may indicate abnormal behaviour in a system.

- In the case of detecting botnets, selecting the right location to place our logging correlation algorithm is an important issue. Another problem is that we do not know how many hosts should be monitored? Monitoring all hosts would be an inefficient approach because we will have large amount of data to process. If we choose n hosts to monitor, the location of these hosts will be critical. In addition, the log files can be discovered by the attacker. One solution to this problem is to encrypt these log files.

- The collected data is passed to Spearman's rank correlation algorithm or DCA for analysis. The DCA could be placed either on the running host or in a demilitarized zone (DMZ) which will be protected from attacks.

- In the case of Spearman's rank correlation, having a large number of idle periods increases the correlation factor, thus generating a large number of false positive alarms.

- One of the most important issue when using the DCA is the selection of signals. Choosing the right signals increases the performance of detection while wrong selection of signals can generate false alarms. Another important issue is the selection of signal weight values as we have seen previously. These weight values have large impact on the generated results.

- The attacker can evade our monitoring system by making a bot very stealthy. The attacker can allow a bot to respond to the issued commands in different time windows. In addition, the attacker can make the bot send a small amount of data at different times. Furthermore, the attacker can implement a bot to perform a stealthy SYN attack, UDP attack or ICMP attack, thus, affecting the correlation of activities and the performance of Spearman's rank correlation and the DCA.

8.4 Future Work

For future research, the above critical points should be taken into account to improve our detection framework. For example, the monitoring program can be implemented to work in a kernel-land rather than in a user-land which provides a better protection level to our monitoring program by hiding the program from the attacker. In addition, by implementing the program in kernel-land, the administrator has more control on monitoring user activities.

One suggestion for the location of the botnet detection framework, is to place the correlation algorithm in a demilitarized zone. This provides a high level of protection from being discovered by the attacker.

As mentioned above, one of the most important issue in the DCA is the proper selection of signals. Currently, the selection of signals is based on pre-observation of bots. For future work, it will be a very useful task if we could implement a monitoring program which has a range of pre-defined signals and the choice of dynamically selecting the most effective signal. In addition, we examined the process of normalization of signals. This process can be further improved by finding a standard normalization procedure for signals by using logarithmic scale as we did in detecting P2P bots (Chapter 7). As a result, more security application areas with different experiments should be conducted to see the affect of choosing logarithmic scale to normalize the signals.

For P2P bots which represents the upcoming threat in the area of botnet detection, we plan to analyse more P2P bots and see if the DCA can detect these bots. New PAMP or danger signals can be used in this area such as the rate of the number of connections generated per second with anomaly score factor taking an advantage of bots trying to connect to many peers in the network.

In addition, new command and control structure bots such as hybrid bots should be examined using DCA. These bots will have advanced features which make them hard to detect. These bots are stealthier and randomly respond to attacker commands in different time periods, introduce traffic which is similar to normal traffic and apply polymorphism techniques to change the bot signatures and behaviours. In this case, the need to adapt proper input signals is required instead of manual selection.

A combination of host-based and network-based bot detection can improve the detection performance. In our future work, we intend to design a framework that combines the evidence of host-based malicious behaviours with the evidence of network-based malicious behaviours and examines their relationship by analysing network traffic, applying some kind of filtering techniques (e.g. white-listing) or looking for well known bots signatures. Furthermore, we want to investigate the area of detecting bots in other operating systems such as Unix/Linux operating systems and Mac operating systems.

The final step is to look for bots mitigation and defenses. Once the bot has been detected on a host, what kind of actions should be taken to mitigate the affect of the bot and how to respond to this threat. We plan to examine this area in our future work.

Bibliography

[1] U. Aickelin, P. Bentley, S. Cayzer, J. Kim, and J. McLeod. *Danger theory: The link between AIS and IDS. In Proc. of the 2nd International Conference on Artificial Immune Systems (ICARIS).* LNCS 2787, pages 147-155. Springer-Verlag, 2003.

[2] API for hackers. `http://sysspider.vectorstar.net/papers/api4hackers.txt`.

[3] Y. Al-Hammadi, and C. Leckie. *Anomaly Detection for Internet Worms.* In Proceeding to the 9th IFIP/IEEE International Symposium on Integrated Network Management. pp 133-146, Nice, France, May 15-19, 2005.

[4] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. *State of the Practice of Intrusion Detection Technologies.* CMU/SEI Technical Report (CMU/SEI-99-TR-028). 1999.

[5] I. Arce, and E. Levy. *An Analysis of the Slapper Worm.* Proceeding of the IEEE Symposium on the Research in Security and Privacy, 2003.

[6] M. Aslam, R. N. Idrees, M. M. Baig and M. A. Arshad. *Anti-Hook Shield against the Software Key Loggers.* Proc. of Nat. Conf. of Emerging Technologies. pp 189191. 2004.

[7] J. Balthrop. *ROIT: A responsive system for mitigating computer network epidemics and attacks.* Master's thesis, University of Mexico, 2005.

[8] G. Bancroft and G. O'Sullivan, *MATHS AND STATISTICS FOR ACCOUNTING*

AND BUSINESS STUDIES, 2^nd ed. Published by McGRAW-HILL Book Company (UK) Limited. pp 135–139. 1988.

[9] P. Bacher, T. Holz, M. Kotter and G. Wicherski. *Know your enemy: Tracking Botnets. Using honeynets to learn more about Bots.* The Honeynet Project and Research Alliance. March 13th 2005). `http://www.honeynet.org/papers/bots/`.

[10] N. Banglamung. *Combination of Misuse and Anomaly Network Intrusion.* Version 1.0. Kaleton Internet, PO Box 188 Shipley BD17 5WU United Kingdom. March 2002.

[11] M. B. Barcena and A. Pesoli. *The new iBotnet.* Virus Bulletin, April 2009. `http://www.virusbtn.com/virusbulletin/archive/2009/04/vb200904-ibotnet`.

[12] P. Barford and V. Yegneswaran. *An Inside Look at Botnets.* Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.

[13] R. Baumann and C. Plattner. *White Paper: Honeypots.* February 2002.

[14] R. Bejtlich *Extrusion Detection: Security Monitoring for Internal Intrusions.* Addison-Wesley; November 2005, ISBN 0321349962.

[15] H. Bersini and F. Varela. *Hints for adaptive problem solving gleaned from immune network.* In Parallel Problem Solving from Nature, pages 343 - 354. Springer-Verlag, 1991.

[16] J. R. Binkley and S. Singh. *An algorithm for anomaly-based botnet detection.* In Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI), pages 4348, July 2006.

[17] R. D. Boer, A. Perelson, and I. Kevrekidis. *Immune network behaviour: From stationary states to limit cycle oscilations.* Bulletin of Mathematical Biology, 55(4):745-780, 1993.

[18] S-D. Bolboaca and L. Jantschi. *Pearson versus Spearman, Kendall's Tau Correlation Analsis on Structure-Activity Relationships of Biological Active Compounds.* Leonardo J. Sci. 2006, 9, 179-200.

[19] J. Bolliger, and T. Kaufmann. *Detecting Bots in Internet Relay Chat Systems.* Semester Thesis SA-2004.29. Swiss Federal Institute of Technology Zurich. 2004.

[20] F. Burnet, G. Bell, A. Perelson, and G. Pimbley. *Theoretical Immunology*, pages 63-85. Marcel Dekker Inc, 1978.

[21] J. Butler and S. Sparks. *Windows Rootkits of 2005.* Parts 1-3. `http://www.securityfocus.com/infocus/1850`.

[22] L. d. Castro and F. V. Zuben. *The clonal selection algorithm with engineering applications.* In Proc. of the Genetic and Evolutionary Computation Conference (GECCO), Workshop on Artificial Immune Systems, pages 36-37. Morgan Kaufmann, 2000.

[23] L. R. d. Castro and J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Paradigm.* Springer-Verlag, London UK, September 2002.

[24] N. Chatzis. *Motivation for behaviourbased dns security: A taxonomy of dnsrelated internet threats.* SECURWARE 2007: Proc. of the International Conference on Emerging Security Information, Systems, and Technologies. Los Alamitos, CA, USA: IEEE Computer Society, pp. 36–41, 2007.

[25] I. R. Cohen. *Tending Adam's Garden : Evolving the Cognitive Immune Self.* Academic Press, 2004.

[26] E. Cooke, F. Jahanian, and D. McPherson. *The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets.* TUGboat Volume 9, Issue 1, 1988.

[27] A. Coutinho. *The Le Douarin phenomenon: a shift in the paradigm of developmental self tolerance.* International Journal of Developmental Biology, 49(2-3):131-136, 2005.

[28] W. Cui, R. Katz, and W. Tan. *BINDER: An Extrusion-based Break-in Detector for Personal Computers.* In Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC), Tuscon, AZ, December 2005.

[29] V. Cutello, G. Nicosia, M. Pavone, and J. Timmis. *An immune algorithm for protein structure prediction on lattice models.* IEEE Transactions on Evolutionary Computation, 11(1):101-117, 2007.

[30] D. Dagon, G. Gu, C. Lee, and W. Lee. *A taxonomy of botnet structures.* In Proceedings of the 23 Annual Computer Security Applications Conference (ACSAC07), 2007.

[31] D. Dagon, C. Zou, and W. Lee. *Modeling Botnet Propagation Using Time Zones.* In Proceedings of the 13th Network and Distributed System Security Symposium NDSS, February 2006.

[32] H. Debar, M. Dacier, and A. Wepsi. *A Revised Taxonomy for Intrusion-Detection Systems.* IBM Research Report. 1999.

[33] D. Dittrich and S. Dietrich. *P2P as botnet command and control: a deeper insight.* In Proceedings of the 2008 3rd International Conference on Malicious and Unwanted Software - Malware, October 2008.

[34] G. Erdelyi. *Hide'n'Seek - Anatomy of Stealth Malware.* Blackhat Briefings, Europe, 2004.

[35] J. Farmer, N. Packard, and A. Perelson. The immune system, adaptation and machine learning. Physica D, 2(1-3):187-204, 1986.

[36] *Firefox web browser.* `http://www.mozilla.com/firefox/`.

[37] E. Florio and M. Ciubotariu. *Peerbot: Catch me if you can.* Whitepaper: Symantec Security Response, Ireland. Originally published by Virus Bulletin, March 2007.

[38] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. *A sense of self for unix processes.* In Proc. of the IEEE Symposium on Research in Security and Privacy, pages 120-128. IEEE Computer Society Press, 1996.

[39] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri. *Self-nonself discrimination in a computer.* In Proc. of the IEEE Symposium on Security and Privacy, pages 202-209. IEEE Computer Society, 1994.

[40] F. C. Freiling, T. Holz, and G. Wicherski. *Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks.* In proceedings to the European Symposium on Research in Computer Security (ESORICS). pages 319-335, Milan, Italy, September 12-14, 2005.

[41] P. Garg. *StraceNT - A System Call Tracer for Windows.* `http://www.intellectualheaven.com/default.asp?BH=projects&H=strace.htm`.

[42] A. Ghizzoni. *Method for Injecting Code into Another Process.* Patent No.: US6698016 B1, Feb 2004.

[43] Gnutella Protocol. *How Gnutella Works?.* `http://computer.howstuffworks.com/file-sharing1.htm`.

[44] J. Goebel and T. Holz. *Rishi: Identify bot contaminated hosts by irc nickname evaluation.* In USENIX Workshop on Hot Topics in Understanding Botnets (HotBots'07), 2007.

[45] J. Govil and J. Govil. *Criminology of BotNets and their Detection and Defense Methods.* 2007 IEEE Electro/Information Technology Conference (EIT07), Chicago, USA, May 2007.

[46] J. Greensmith. *The Dendritic Cell Algorithm.* PhD thesis, School of Computer Science, University Of Nottingham, 2007.

[47] J. Greensmith et al. *Interdisciplinary Perspective.* Journal paper, currently in preparation, 2009.

[48] J. Greensmith and U. Aickelin. *Artificial Dendritic Cells: Multi-faceted Perspectives.* Book chapter on Human-Centric Information Processing Through Granular Modelling, pp. 375-395, 2009.

[49] J. Greensmith and U. Aickelin. *Dendritic cells for syn scan detection.* In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007), pages 49-56, 2007.

[50] J. Greensmith and U. Aickelin. *The deterministic dendritic cell algorithm.* In Proc. of the 7th International Conference on Artificial Immune Systems (ICARIS), pages 291-302. Springer, 2008.

[51] J. Greensmith, U. Aickelin, S. Cayzer. *Introducing dendritic cells as a novel immune-inspired algorithm for anomaly detection, in Proceedings of the Fourth International Conference on Artificial Immune Systems (ICARIS-05).* LNCS 3627, pp. 153-167, 2005.

[52] J. Greensmith, U. Aickelin and G. Tedesco, *Information Fusion for Anomaly Detection with the Dendritic Cell Algorithm.* Special Issue on Biologically Inspired Information Fusion; International Journal of Information Fusion, Elservier, 2008.

[53] J. Greensmith, U. Aickelin, and J. Twycross. *Articulation and clarification of the Dendritic Cell Algorithm.* In Proc. of the 5th International Conference on Artificial Immune Systems (ICARIS), LNCS 4163, pages 404-417, 2006.

[54] J. Greensmith, U. Aickelin, and J. Feyereisl. *The DCA-SOMe comparison: A comparative study between two biologically-inspired algorithms.* Evolutionary Intelligence: Special Issue on Artificial Immune Systems, accepted for publication, 2008.

[55] J. Greensmith, A. Whitbrook and U. Aickelin. *Artificial Immune Systems.* Handbook of Metaheuristics, 2nd edition, Springer, 2010.

[56] F. Gu, J. Greensmith, W. Chen and U. Aickelin. *Theoretical Aspects of the Dendritic Cell Algorithm.*`http://ima.ac.uk/slides/fxg-17-11-2009.pdf`.

[57] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. *Peer-to-peer botnets: Overview and case study.* In HotBots 07 conference, 2007.

[58] F. Gu, J. Greensmith and U. Aickelin. *Further Exploration of the Dendritic Cell Algorithm: Antigen Multiplier and Time Windows.* Proceedings of the 7th International Conference on Artificial Immune Systems (ICARIS 2008), Phuket, Thailand, pp. 142-153, 2008.

[59] G. Gu, R. Perdisci, J. Zhang, and W. Lee. *Botminer: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection.* In USENIX Security, 2008.

[60] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. *BotHunter: Detecting Malware Infection through IDS-driven Dialog Correlation.* In 16th USENIX Security Symposium, 2007.

[61] G. Gu, J. Zhang, and W. Lee. *Botsniffer: Detecting Botnet Command and Control Channels in Network Traffic.* In NDSS, 2008.

[62] C. Hank. *DLL Injection and function interception tutorial.* `http://www.codeproject.com/KB/DLL/DLL_Injection_tutorial.aspx`.

[63] C. Herley and D. Florencio. *How To Login From an Internet Cafe Without Worrying About Keyloggers*, Symposium on Usable Privacy and Security (SOUPS) 06. July 2006.

[64] S. Herman. *The McAfee Network Access Control Solution Protect Your Network from the Risk of non-Compliant Systems.* McAfee. `http://unatekconference.com/images/pdfs/presentations/Berkuta.pdf`.

[65] M. M. Hoff. *Botnets - a cancer on Internet.* Leipzig University of Applied Science. (January 2007).

[66] S. Hofmeyr. *An immunological model of distributed detection and its application to computer security.* PhD thesis, University Of New Mexico, 1999.

[67] S. Hofmeyr and S. Forrest. *Immunity by design.* In Proc. of the Genetic and Evolutionary Computation Conference (GECCO), pages 1289-1296, 1999.

[68] G. Hoglund and J. Butler. *Rootkits : Subverting the Windows Kernel.* Addison-Wesley Professional. July 22, (2005).

[69] T. Holz. *A Short Visit to the Bot Zoo.* Journal of IEEE Security & Privacy. Vol 3, number 3, pp. 76–79, 2005.

[70] T. Holz and F. Raynal. *Detecting honeypots and other suspicious environments.* In Sixth Annual IEEE Systems, Man and Cybernetics (SMC) Information Assurance Workshop (IAW05), 2005.

[71] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. *Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm.* In LEET 08: First USENIX Workshop on Large-Scale Exploits and Emergent Threats, 2008.

[72] Honeynet Project. *Scan of the Month 25: Slapper Worm .unlock.c source file*, 2002. `http://www.honeynet.org/scans/scan25/.unlock.nl.c`.

[73] *HookAPI Source Code.* Retrieved Jun. 2nd 2006 from `http://www.codeproject.com/system/Paladin.asp`.

[74] E. Hurley. *Slapper worm exploits OpenSSL hole, sets up P2P network.* `http://searchsecurity.techtarget.com/news/article/0,289142,sid14_gci851200,00.html`.

[75] G. Hunt and D. Brubacher. *Detours: Binary Interception of Win32 Functions.* Proceedings of the 3rd USENIX Windows NT Symposium, USENIX, pp. 135-143. Seattle, WA, July 1999.

[76] N. Ianelli and A. Hackworth. *Botnets as a Vehicle for Online Crime.* CERT Coordination Center, 2005.

[77] *IceChat - IRC Client.* `http://www.icechat.net/`. Accessed on 10th December 2008.

[78] Internet Security Systems. *Network- vs. Host-based Intrusion Detection: A Guide to Intrusion Detection Technology.* 6600 Peachtree-Dunwoody Road, 300 Embassy Row, Atlanta, GA 30348.

[79] IRC Protocol. `http://www.rfceditor.org`.

[80] F. Jahanian. *Enter the Botnet: An Introduction to the Post-Worm Era.* ARO-DARPA-DHS Special Workshop on Botnets, 2006.

[81] C. Janeway. *Approaching the asymptote? Evolution and revolution in immunology.* Cold Spring Harbor Symposium on Quant Biology, 1:1-13, 1989.

[82] R. Jeffrey. *Load Your 32-bit DLL into Another Process's Address Space Using INJLIB.* Microsoft Systems Journal Volume 9 Number 5. May 1994.

[83] Z. Ji and D. Dasgupta. *Revisiting negative selection algorithms.* Evolutionary Computation,15(2):223-251, 2007.

[84] Jones and Jim. *BotNets: Detection and Mitigation.* FEDCIRC, February 2003.

[85] P. Kanwar. *The Art Of Key logging - Implementation and Detection On Windows Platform.* `http://www.hellboundhackers.org/articles/155-the-art-of-keylogging-implementation-and-detection.html`

[86] A. Karasaridis, B. Rexroad and D. Hoeflin. *Wide-Scale Botnet Detection and Characterization.* 1st Workshop on Hot Topics in Understanding Botnets, April 2007.

[87] P. Karbhari, M. Ammar, A. Dhamdhere, H. Raj, G. Riley, and E. Zegura. *Bootstrapping in Gnutella: A Measurement Study.* In PAM, April 2004.

[88] Y. Kaplan. *API spying techniques for windows 9x, NT, and 2000.* (April 2000). `http://www.internals.com/ar-ticles/apispy/apispy.htm`.

[89] J. Kephart, G. Sorkin, M.Swimmer, and S. White. *Blueprint for a computer immune system.* In Virus Bulletin International Conference, 1999.

[90] J. Kim and P. Bentley. *Towards an Artificial. Immune System for Network Intrusion Detection: An investigation of dynamic clonal selection with a negative selection operator.* In Proc. Of the Congress of Evolutionary Computation (CEC2001), pp. 1244-1252, 2002.

[91] J. Kim and P. Bentley. *Evaluating negative selection in an artificial immune system for network intrusion detection.* In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), ACM Press, San Francisco, USA. pp. 1330-1337. July 2001.

[92] S. Kim. *Intercepting System API Calls.* Retrieved Jun. 2^{nd} 2006 from `http://cache-www.intel.com/cd/00/00/21/70/217023_217023.pdf`. May 2004.

[93] Know Your Enemy: Honeynets. *What a honeynet is, its value, overview of how it works, and risk/issues involved.* The Honeynet Project. May 2006. `http://old.honeynet.org/papers/honeynet/`.

[94] S. Kondo and N. Sato. *Botnet Traffic Detection Techniques by C&C Session Classification using SVM.* In Proceedings of the Second International Workshop on Security (IWSEC 2007), Nara, Japan, pp. 91104, October 29-31, 2007.

[95] H. Kozushko. *Intrusion Detection: Host-Based and Network-Based Intrusion Detection Systems.* Thursday, September 11, 2003.

[96] S. Kumar and E. H. Spafford. *A Pattern Matching Model for Misuse Intrusion Detection.* Proceeding of the 17th National Computer Security Conference. Purdue University, 1994.

[97] R. Kuster. *Three ways to Inject Your Code into Another Process.* Retrieved Jun. 2nd 2006 from `http://www.codeguru.com/Cpp/W-P/system/processesmodules/article.php/c5767`.

[98] J. Levine et al. *The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks.* Proceedings of the 2003 IEEE Workshop on Information Assurance, pp 92-99, June 2003.

[99] C. Livadas, R. Walsh, D. Lapsley and W. T. Strayer. *Using machine learning techniques to identify botnet traffic.* In Proceedings of the 2nd IEEE LCN Workshop on Network Security (WoNS2006), 2006.

[100] J. W. Lockwood et al. *Application of Hardware Accelerated Extensible Network Nodes for Internet Worm and Virus Protection.* Washington University, and Global Velocity. St. Louis, USA. Nov. 2003.

[101] M. Lutz and G. Schuler. *Immature, semi-mature and fully mature dendritic cells: which signals induce tolerance or immunity?.* Trends in Immunology, 23(9): 991-1045, 2002.

[102] P. Matzinger. *Tolerance, danger and the extended family.* Annual Reviews in Immunology. 12:991-1045, 1994.

[103] P. Maymounkov and D. Mazi'eres, *Kademlia: A peer-to-peer information system based on the XOR metric.* In 1st International Workshop on Peer-to-Peer Systems, pp. 5362, (March 2002).

[104] McAfee Avert Labs. *McAfee Threats Report: Second Quarter 2009.* 2009. `http://www.mcafee.com/us/local_content/reports/6623rpt_avert_threat_0709.pdf`.

[105] MessageLabs Intillegence Reports for Q3/September 2009. `http://www.messagelabs.com/mlireport/MLI_2009.09_Sept_SHSFINAL_EN.pdf`

[106] mIRC client application. `http://www.mirc.com`.

[107] D. Moore, C. Shannon, G. Voelker, and S. Savage. *Internet Quarantine: Requirements for Containing Self-Propagating Code.* Proc. IEEE INFOCOM, UC San Diego, California, pp. 1901-1910, 2003.

[108] *MSDN - File Management Functions.* `http://msdn2.microsoft.com/en-us/library/aa364232.aspx`.

[109] *MSDN: Hooks.* http://msdn.microsoft.com/en-us/library/ms632589(VS.85).aspx.

[110] *MSDN - Keyboard Input.* http://msdn2.microsoft.com/en-us/library/ms645530.aspx.

[111] MSDN: SetWindowsHookEx() Function http://msdn.microsoft.com/en-us/library/ms644990(VS.85).aspx.

[112] *MSDN - Winsock Functions.* http://msdn2.microsoft.com/en-us/library/ms741394.aspx.

[113] National Infrastructure Security Co-ordination Centre. *Botnets - the threat to the Critical National Infrastructure.* London, October 17, 2005.

[114] J. Nazario, J. Anderson, R. Wash and C. Connelly. *The Future of Internet Worms.* Crimelabs research, July 20, 2001.

[115] V. Nivargi, M. Bhaowal and T. Lee. *Machine Learning Based Botnet Detection.* CS 229 Final Project Report. Stanford University. 2006.

[116] C. Nunnery and B. B. Kang. *Locating Zombie Nodes and Botmasters in Decentralized Peer-to-Peer Botnets.* Available at: https://www.os3.nl/_media/2007-2008/students/matthew_steggink/rp1/p2pdetect_conceptpaper.pdf?id=2007-2008, (2007).

[117] R. Oates, J. Greensmith, U. Aickelin, J. Garibaldi, and G. Kendall. *The application of a dendritic cell algorithm to a robotic classifier.* In Proc. of the 6th International Conference on Artificial Immune Systems (ICARIS), LNCS 4628, pages 204-215, 2007.

[118] M. Overton. *Bots and Botnets: Risks, Issues and Prevention.* In Proceedings of Virus Bulletin Conference 2005. Dublin, Ireland. October 5-7, 2005.

[119] V. Paxon. *Bro: A System for Detecting Network Intruders in Real Time.* In Proceedings of the 7th USENIX Security Symposium, 1998.

[120] F. Perriot and P. Szor. *An Analysis of the Slapper Worm Exploit.* Symantec White Paper. (2003). http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf.

[121] M. Pietrek. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.* http://msdn.microsoft.com/en-gb/magazine/ms809762.aspx

[122] M. Pietrek. *An In-Depth Look into the Win32 Portable Executable File Format.* http://msdn.microsoft.com/en-gb/magazine/cc301805.aspx.

[123] P. Porras, H. Saidi, and V. Yegneswaran. *A Multi-perspective Analysis of the Storm (Peacomm)Worm.* Technical report, Computer Science Laboratory, SRI International, October 2007.

[124] N. Provos. *Systrace.*

[125] N. Provos. *A Virtual Honeypot Framework.* In Proceedings of the 13th USENIX Security Symposium, pp 1–14, 2004.

[126] S. Racine. *Analysis of Internet Relay Chat Usage by DDoS Zombies.* Master's Thesis. Swiss Federal Institute of Technology Zurich. April 2004.

[127] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. *My botnet is bigger than yours (maybe, better than yours).* In Proceedings of HotBots07, 2007.

[128] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. *A Multifaceted Approach to Understanding the Botnet Phenomenon.* In Proceedings of ACMSIGCOMM/USENIX Internet Measurement Conference (IMC), pages 4152, Oct., 2006.

[129] A. Ramachandran, N. Feamster and D. Dagon. *Revealing botnet membership using DNSBL counter-intelligence.* In Proceedings of USENIX SRUTI06, 2006.

[130] D. Sancho. *The Future of Bot Worms What we can expect from worm authors in the coming months.* Trend Micro Incoperated, 2005.

[131] R. Schoof and R. Koning. *Detecting peer-to-peer botnets.* University of Amsterdam, 2007.

[132] Sdbot05b. Retrieved Jun. 2nd, (2006). http://www.securityforest.com/downloads/bots/sdbot05b.zip.

[133] R. Sharp. *An Introduction to Malware.* Spring 2008.

[134] C. Shetty. *Introduction to Spyware Keylogger.* Last retrieved on 12-02-2009. http://www.securityfocus.com/infocus/1829.

[135] A. Silverstein. *Cellular versus humoral immunology: a century-long dispute.* Nature Immunology, 4(5):425-428, 2003.

[136] A. Silverstein. Paul Ehrlich. *Archives and the history of immunology.* Nature Immunology, 6(7):639-639, 2005.

[137] R. Smith, S. Forrest, and A. Perelson. *Searching for diverse, cooperative subpopulations with Genetic Algorithms.* Evolutionary Computation, 1(2):127-149, 1993.

[138] SNORT Intrusion Detection and Prevention System. http://www.snort.org.

[139] E. H. Spafford. *The Internet Worm Program: An Analysis.* Purdue Technical Report. Department of Computer Sciences, Purdue University, West Lafayatte. IN, November 29, 1988.

[140] Spamhous. http://www.spamhaus.org.

[141] L. Spitzner. *Honeypots: Tracking Hackers.* Addison-Wesley Pub Co. 1st edition, ISBN: 0321108957, 2002.

[142] M. Steggink and I. Idziejczac. *Detection of Peer-to-Peer Botnets.* University of Amsterdam. 2007-2008.

[143] J. Stewart. *Phatbot Trojan Analysis.* http://www.secureworks.com/research/threats/phatbot/.

[144] J. Stewart. *Sinit P2P Trojan Analysis.* December 8, 2003. http://www.secureworks.com/research/threats/sinit/.

[145] J. Stewart. *Storm Worm DDoS Attack.* `http://www.secureworks.com/research/threats/view.html?threat=storm-worm`. Feb 2007.

[146] J. Stewart. *Spam Botnets to Watch in 2009.* `http://www.secureworks.com/research/threats/botnets2009/?threat=botnets2009`. Jan 13, 2009.

[147] T. Stibor , P. Mohr , J. Timmis , C. Eckert. *Is negative selection appropriate for anomaly detection?.* Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC, USA, June 25-29, 2005.

[148] T. Stibor, J. Timmis and E. Claudia. *A Comparative Study of real-valued negative selection to statistical anomaly detection techniques.* In Proceedings of the 4th International Conference on Artificial Immune Systems (ICARIS-2005), Banff, Alberta, Canada, August 14-17, 2005.

[149] T. Stibor, J. Timmis, and C. Eckert. *On permutation masks in hamming negative selection.* In Proceedings of 5th International Conference on Artificial Immune Systems (ICARIS), Lecture Notes in Computer Science, pages 122–135. Springer-Verlag, 2006.

[150] T. Stibor, J. Timmis, and C. Eckert. *Artificial Immune Systems for IT-security.* IT-Information Technology, 48(3), 2006.

[151] E. Stinson, J. C. Mitchell. *Characterizing Bots Remote Control behaviour.* In Detection of Intrusions & Malware, and Vulnerability Assessment, July 2007.

[152] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. *Analysis of the Storm and Nugache Trojans: P2P is here.* In USENIX ;login: vol. 32, no. 6, December 2007. `http://www.usenix.org/`.

[153] W. T. Strayer, R. Walsh, C. Livadas, D. Lapsley, *Detecting Botnets with Tight Command and Control.* Proceedings to the 31st IEEE Conference on Local Computer Networks, lcn, pp. 195-202, 2006.

[154] W. T. Strayer, D. Lapsley, R. Walsh and C. Livadas. *Botnet Detection Based on Network Behavior*. Book chapter of *Botnet Detection: Countering the Largest Security Threat*. Publisher:Springer, ISBN 978-0-387-68768-1, Vol. 36, 2008.

[155] S. Subramanyam, C. E. Frank, D. H. Galli. *Keyloggers: The Overlooked Threat to Computer Security.* First Midstates Conference for Undergraduate Research in Computer Science and Mathematics, Department of Mathematics and Computer Science, Denison University, Granville, Ohio, 2003.

[156] C. K. TAN. *Windows Key Logging and Counter-Measure.*

[157] I. Taylor. *Gnutella.* Lecture Notes in Distributed Systems, 2003.

[158] TCPDUMP `http://www.tcpdump.org`.

[159] *The Evolution of Malicious IRC bots.* White Paper: Symantec Security Response. In proceedings of the VB2005 Conference. 2005.

[160] The Overnet Protocol. OpenSVN. `https://opensvn.csie.org/mlnet/trunk/docs/overnet.txt`.

[161] *The Weakness of the Windows API.* Part 1 in a 3 Part Series. Version 1.0. October 2005.

[162] V. L. L. Thing, M. Sloman, and N. Dulay. *A survey of bots used for distributed denial of service attacks.* In Proceedings to the 22nd IFIP International Information Security Conference (SEC 07). Sandton, Gauteng, South Africa, 2007.

[163] J. Timmis. *Artificial immune systems: today and tomorrow.* Natural Computing, 6(1):1-18, March 2007.

[164] Trend Micro. *The Future of Bot Worms.* (2005).

[165] Trend Micro. *Taxonomy of botnet threats.* A Trend Micro White Paper, 2006.

[166] J. Twycross. *Integrated Innate and Adaptive Artificial Immune Systems Applied to Process Anomaly Detection.* PhD thesis, University Of Nottingham, 2007.

[167] J. Twycross and U. Aickelin. *Libtissue - implementing innate immunity.* In Proc. of the Congress on Evolutionary Computation (CEC), pages 499-506, 2006.

[168] Virus Bulletin. *The International Publication on Computer Virus Prevention, Recognition and Removal.* August 2003.

[169] X. Wang, Z. Li, N. Li and J. Choi. *PRECIP: Towards Practical and Retrofittable Confidential Information Protection.* In Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), 2008.

[170] Y. M. Wang, B. Vo, R. Roussev, C. Verbowski and A. Johnson. *Strider Ghost-Buster: Why Its A Bad Idea For Stealth Software To Hide Files.* Technical Report MSR-TR-2004-71. Microsoft Research, Redmond, USA. July 24, 2004.

[171] P. Wang, S. Sparks, and C. Zou. *An advanced hybrid peer-to-peer botnet.* In First Workshop on Hot Topics in Understanding Botnets, 2007.

[172] *WASTE Project.* `http://waste.sourceforge.net/`.

[173] What is WASTE? `http://files.minuslab.net/waste%20design.doc`

[174] C. Williams, R. Harry and J. McLeod. *Mechanism of apoptosis induced DC suppression.* Journal of Immunology, 2007.

[175] *Win32 Portable Userland Rootkit.* `http://www.phrack.com/issues.html?issue=62&id=12`.

[176] W. Yang,B-X. Fang, B. Liu, and H-L Zhang. *Intrusion detection system for high-speed network.* Journal of Computer Communications. Volume 27, Issue 13, pp. 1288-1294, August 2004.

[177] Z. Yonglin *Botnets - Threats and Countermeasures.* Jakarta, August 2005.

[178] C. C. Zou and R. Cunningham. *Honeypot-aware advanced botnet construction and maintenance.* In International Conference on Dependable Systems and Networks (DSN06), 2006.

[179] C. C. Zou et al. *Monitoring and Early Warning for Internet Worms.* University of Massachusetts at Amherst. Washington, DC, USA. October 27-30, 2003.

Appendix A

Publications

A.1 Conference Papers:

- Yousof Al-Hammadi, Uwe Aickelin. *Behavioural Correlation for Detecting P2P Bots*. Proceedings of the Second International Conference on Future Networks (ICFN 2010), Sanya, Hainan, China, pp. 323-327, 2010.

- Yousof Al-Hammadi, Uwe Aickelin and Julie Greensmith. *DCA for Bot Detection*. Proceedings of the IEEE World Congress on Computational Intelligence (WCCI2008), Hong Kong, pp. 1807-1816, (June 2008).

- Yousof Al-Hammadi and Uwe Aickelin. *Detecting Bots Based on Keylogging Activities*. Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES2008), Barcelona , Spain, pp. 896-902, (March 2008).

- Yousof Al-Hammadi and Uwe Aickelin. *Detecting Botnets Through Log Correlation*. Proceedings of the Workshop on Monitoring, Attack Detection and Mitigation (MonAM2006), Tuebingen, Germany, pp. 97-100 (September 2006).

- Y. Al-Hammadi and C. Leckie. *Anomaly Detection for Internet Worms*. In Proceedings of the Ninth IEEE International Symposium on Integrated Network Management (IM 2005), Nice, France, pp. 133-146. (May 2005).

A.2 Journal Paper

- Yousof Al-Hammadi, Uwe Aickelin, Julie Greensmith. *Performance Evaluation of DCA and SRC on a Single Bot Detection.* Journal of Information Assurance and Security, 5 (1), pp. 265-275, 2010.

- Yousof Al-Hammadi, Uwe Aickelin and Julie Greensmith. *Dendritic Cell Algorithm for Detecting Peer-to-Peer Bots.* Submitted to: Special Issue of Computer Communications on Building Secure Parallel and Distributed Networks and Systems.

Appendix B

Glossary

Table B.1: Glossary A.

Symbol	*Definitions*
API	Application Programming Interface. Set of routines and and tools which help an application to request services from the operating system.
AOL	America On Line. An online service.
AOP	Approved Channel Operator.
BBS	Bulletin Board System. An electronic message center.
C&C	Command and Control A structure used to send commands by the attacker and receive response from the bots.
CGI	Common Gateway Interface. A specification for transferring information between webservers and CGI programs.
DCA	Dendritic Cell Algorithm is a multi-sensor data fusion on a set of input signals and these information are correlated with potentially anomalous 'suspect entities'.
DCC	Direct Client to Client which enables uses to communicate directly with each other.
DCOM	Distributed Component Object Model. An interface to allow components to communicate over the network.
DDoS	Distributed Denial of Service attack. An attack carried out when multiple hosts infected by trojan viruses consume the bandwidth of a network or a system by generating large number of packets causing a denial of service.
DLL	Dynamic Link Library. A library for shared executable functions between Windows applications.
DMZ	A DeMilitarized Zone, a protected network which sits between a internal and external network.
DNS	Domain Name Server. A service which responsible for translating domain names into numeric IP addresses.
DNSBL	DNS Black List.
DS	Danger Signal. A measure of an attribute which increases in value to indicate deviation from usual behaviour. Low values of this signal may not be anomalous, giving a high value confidence of indicating abnormality. PAMP signal has less effect on the output signal than DS signal.

Table B.2: Glossary B.

Symbol	*Definitions*
DU	Destination Unreachable is an ICMP error message.
FTP	File Transfer Protocol used to transfer files between computers.
HIDS	Host-based Intrusion Detection Systems.
HTTP	Hyper Text Transfer Protocol used by World Wide Web (WWW) to communicate with webservers and browsers.
IAT	Import Address Table.
ICMP	Internet Control Message Protocol for network troubleshooting and location.
IDS	Intrusion Detection Systems used to inspect inbound and outbound network packets and generate alarms if suspicious activity has been detected.
IM	Instant messaging.
IP	Internet Protocol. Network layer protocol which specifies the format of packets.
ISP	Internet Service Provider.
IRC	Internet Relay Chat. A special protocol for real-time chat. The botmasters use this protocol to control their bots.
LSASS	Local Security Authority Subsystem Service.
MAC	MCAV Anomaly Context value is a modification of the MCAV.
MCAV	Mature Context Anomaly Value. A value between 0 and 1 to measure the abnormality of a process.
MSSQL	Microsoft Structured Query Language.
MUT	A chat program that allows user to talk with each other.
NIDS	Network-based Intrusion Detection System.
PAMP	Pathogen Associated Molecular Patterns A strong evidence of abnormal/bad behaviour. An increase in this signal is associated with a high confidence of abnormality.
PC	Personal Computer.
PE	Portable Executable is a file format for executables and DLLs, used in Windows operating systems.

Table B.3: Glossary C.

Symbol	*Definitions*
RFC	Request for Comments which provides information about the Internet.
RPC	Remote Procedure Call provides ability to execute code remotely.
SMB	Server Message Block which is a protocol for sharing files, printers and serial ports between computers.
SRC	Spearman's Rank Correlation algorithm defines the correlation value between two datasets.
SS	Safe Signal. A measure which increases value in conjunction observed normal behaviour. This is a confident with indicator of normal, predictable or steady-state system behaviour. This signal is used to counteract the effects of PAMP and DS signals and thus has negative impact on the output signals.
TCL	Tool Command Language. A scripting language.
TFTP	Trivial File Transfer Protocol. A simple form of FTP.
TTL	Time-To-Live, a field in IP which specify how many hops a packet can visits before being discarded.
UDP	User Datagram Protocol. A connectionless protocol which provides few error recovery services in comparison to TCP.
VM	Virtual Machine. A self-contained environment which runs on a physical computer often used for security purposes.

Appendix C

Hooking Techniques and Steps

C.1 System-wide Hook Types

- WH_CALLWNDPROC and WH_CALLWNDPROCRET: monitor messages sent to window procedures.
- WH_CBT: intercepts messages before activating, destroying, minimizing, maximizing, moving or resizing a window.
- WH_DEBUG: hooks the existing hook. It is called before another system hook is used.
- WH_FOREGROUNDIDLE: allows the user to execute low-priority tasks when its foreground thread is idle.
- WH_GETMESSAGE: monitors messages to be returned by GetMessage() and PeakMessage().
- WH_JOURNALPLAYBACK: inserts messages into the system message queue.
- WH_JOURNALRECORD: monitors and records input events.
- WH_KEYBOARD_LL: monitors keyboard input events about to be posted in some input queue.
- WH_KEYBOARD: monitors WM_KEYDOWN and WM_KEYUP which indicates the state of a key.

- WH_MOUSE_LL: monitors mouse input events about to be posted in some input queue.

- WH_MOUSE: monitors a mouse message which is about to be returned by GetMessage() and PeakMessage().

- WH_MSGFILTER and WH_SYSMSGFILTER: is called when messages are about to be processed by a menu, scroll bar, message box, or dialog box.

- WH_SHELL: used to receive important notices.

C.2 Hooking Steps by Manipulating modules IAT

1. Find the address of the function you want to hook (for example, Sleep() function which reside inside kernel32.dll) by using:

```
HMODULE hMod = GetModuleHandle(”kernel32.dll”);
PROC AddrOfOrigfunc=GetProcAddress(hMod,”Sleep”);
```

2. Find the handle of the program that call the hook function by:

```
HMODULE hInstance = GetModuleHandle(NULL); // or specify the name of the program (e.g. myprog.exe)
```

3. Define the required variables:

```
PROC MyHookedFunction = (PROC) MySleep; //get the address of our replaced function
PROC *ppfn; //to store the address of Sleep once we find it
```

4. Call PIMAGE_IMPORT_DESCRIPTOR and ImageDirectoryEntryToData to find the IAT of the program (myprog.exe) // pImportD is the entry of IAT:

```
PIMAGE_IMPORT_DESCRIPTOR pImportD =
ImageDirectoryEntryToData(hInstance, TRUE, IMAGE_DIRECTORY_
ENTRY_IAT,&ulSize);
if (pImportD == NULL) then
```

```
    return; //This module has no import section
end if
```

5. Loop through all the modules (user32.dll, kernel32.dll, xxx.dll) to find the required module (in our case, we are searching for kernel32.dll):

```
PSTR pszModuleName;
for ( ; pImportD → Name; pImportD++) do
    pszModuleName = = (PSTR) ((PBYTE) hInstance + pImportD → Name);
    if (!strcmp(pszModuleName,“kernel32.dll”)) then
        break;
    end if
end for
if (pImportD == 0) then
    return; // no imported functions have been found in this module
end if
```

6. Retrieve the address of the current function using pThunk:

```
PIMAGE_THUNK_DATA pThunk =
(PIMAGE_THUNK_DATA) ((PBYTE) hInstance + pImportD → FirstThunk);
```

7. Process for replacing current function address with new function address:

```
MEMORY_BASIC_INFO mbi;
// we have our function address previously, so loop through all the function’s
// addresses until our function address (address of Sleep) is matched
for ( ; pThunk → u1.Function; pThunk++) do
    //get the address of the function address
    ppfn = (PROC*) &pThunk → u1.Function;
    bFound = (*ppfn == AddrOfOrigfunc);
    if (bFound) then
```

```
        VirtualQuery(ppfn, &mbi, sizeof(MEMORY_BASIC_INFO));
        VirtualAlloc(mbi.BaseAddress, mbi.ReginSize, MEM_COMMIT,
        PAGE_READWRITE);
        *ppfn = *MyHookedFunction; // the address we found is replaced with our function address
        VirtualAlloc(mbi.BaseAddress, mbi.ReginSize, mbi.State, mbi.Protect);

        break;
    end if
end for
```

C.3 Steps for hooking external process and DLL injection

In order to inject our interception code into another process, the following steps are needed in order to make the external process hooking work [92]:

- Use OpenProcess() function to retrieve a handle to the target process
- Allocate memory in the target process by calling VirtualAllocEx() and write to the allocated memory the name of the DLL to be injected using WriteProcessMemory().
- Get the address of LoadLibrary() which is the same in all processes by calling: GetProcAddress(GetModuleHandle("kernel32.dll", "LoadLibraryW"); This is because the kernel.dll is mapped always to the same address space of every process, therefore, the address of LoadLibrary() function has the same value in address space of any running process.
- Call CreateRemoteThread() to create a remote thread in the target process specifying the entry point of LoadLibrary() and the name of the DLL as its argument. The thread runs LoadLibrary() will load a DLL into the target process.
- Free the allocated memory using VirtualFreeEx().

Appendix D

Signal and Antigen Log File Example

D.1 A Sample of Antigen Log File

```
1240334543.687500 antigen 3472 1 send 2
1240334547.125000 antigen 3472 1 socket
1240334547.250000 antigen 3472 1 connect
1240334547.250000 antigen 3472 1 send 2
1240334548.0 antigen 3472 1 socket
1240334548.93750 antigen 3472 1 connect
1240334548.93750 antigen 3472 1 send 2
1240334548.187500 antigen 3472 1 socket
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegQueryValueW
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyA
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegOpenKeyW
1240334548.250000 antigen 3248 1 RegQueryValueW
1240334548.296875 antigen 3472 1 connect
1240334548.296875 antigen 3472 1 send 2
1240334548.750000 antigen 3472 1 socket
1240334548.750000 antigen 3304 1 ReadFile
```

```
1240334548.765625 antigen 3304 1 send
1240334548.875000 antigen 3472 1 connect
1240334548.875000 antigen 3472 1 send 2
1240334549.265625 antigen 3372 1 send
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegQueryValueW
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyA
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegOpenKeyW
1240334549.578125 antigen 3248 1 RegQueryValueW
1240334550.812500 antigen 3472 1 socket
1240334550.937500 antigen 3472 1 connect
1240334550.937500 antigen 3472 1 send 2
1240334551.171875 antigen 3472 1 send 2
1240334551.421875 antigen 3472 1 socket
1240334551.453125 antigen 3472 1 connect
1240334551.453125 antigen 3472 1 send 2
1240334551.515625 antigen 3472 1 socket
1240334551.546875 antigen 3472 1 connect
1240334551.546875 antigen 3472 1 send 2
1240334551.906250 antigen 3472 1 socket
1240334551.937500 antigen 3472 1 connect
1240334551.937500 antigen 3472 1 send 2
1240334551.984375 antigen 3472 1 socket
1240334552.15625 antigen 3472 1 connect
1240334552.15625 antigen 3472 1 send 2
1240334552.359375 antigen 3472 1 socket
1240334552.531250 antigen 3472 1 connect
1240334552.531250 antigen 3472 1 send 2
1240334552.703125 antigen 3472 1 socket
1240334552.875000 antigen 3472 1 connect
1240334552.875000 antigen 3472 1 send 2
1240334553.375000 antigen 3472 1 socket
1240334553.406250 antigen 3472 1 connect
```

```
1240334553.406250 antigen 3472 1 send 2
1240334554.718750 antigen 3472 1 socket
1240334554.750000 antigen 3472 1 connect
1240334554.750000 antigen 3472 1 send 2
1240334555.125000 antigen 3472 1 socket
1240334555.265625 antigen 3472 1 connect
1240334555.265625 antigen 3472 1 send 2
1240334557.140625 antigen 3304 1 ReadFile
1240334557.140625 antigen 3304 1 send
1240334557.593750 antigen 3472 1 socket
1240334557.718750 antigen 3472 1 connect
1240334557.734375 antigen 3472 1 send 2
1240334559.453125 antigen 3472 1 socket
1240334559.593750 antigen 3472 1 connect
1240334559.593750 antigen 3472 1 send 2
1240334560.812500 antigen 3472 1 socket
1240334560.937500 antigen 3472 1 connect
1240334560.937500 antigen 3472 1 send 2
1240334561.500000 antigen 3472 1 socket
1240334561.593750 antigen 3472 1 connect
1240334561.593750 antigen 3472 1 send 2
1240334562.734375 antigen 3472 1 socket
1240334562.828125 antigen 3472 1 connect
1240334562.828125 antigen 3472 1 send 2
1240334563.296875 antigen 3472 1 socket
1240334563.296875 antigen 3472 1 connect
1240334563.296875 antigen 3472 1 socket
1240334563.296875 antigen 3472 1 connect
1240334563.609375 antigen 3472 1 socket
1240334563.703125 antigen 3472 1 connect
1240334563.703125 antigen 3472 1 send 2
1240334563.812500 antigen 3472 1 socket
1240334563.859375 antigen 3248 1 RegOpenKeyA
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegOpenKeyA
1240334563.859375 antigen 3248 1 RegOpenKeyA
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegQueryValueW
1240334563.859375 antigen 3248 1 RegOpenKeyA
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegOpenKeyA
```

```
1240334563.859375 antigen 3248 1 RegOpenKeyA
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegOpenKeyW
1240334563.859375 antigen 3248 1 RegQueryValueW
1240334563.906250 antigen 3472 1 connect
1240334563.906250 antigen 3472 1 send 2
1240334564.265625 antigen 3372 1 send
1240334564.500000 antigen 3472 1 socket
1240334564.578125 antigen 3472 1 connect
1240334564.578125 antigen 3472 1 send 2
1240334565.734375 antigen 3472 1 socket
1240334565.796875 antigen 3472 1 connect
1240334565.796875 antigen 3472 1 send 2
1240334566.484375 antigen 3472 1 socket
1240334566.593750 antigen 3472 1 connect
1240334566.593750 antigen 3472 1 send 2
1240334568.46875 antigen 3472 1 socket
1240334568.140625 antigen 3472 1 connect
1240334568.140625 antigen 3472 1 send 2
1240334568.328125 antigen 3472 1 send 2
1240334568.578125 antigen 3472 1 socket
1240334568.718750 antigen 3472 1 connect
1240334568.718750 antigen 3472 1 send 2
1240334569.671875 antigen 3472 1 socket
1240334569.796875 antigen 3472 1 connect
1240334569.796875 antigen 3472 1 send 2
1240334570.250000 antigen 3472 1 socket
1240334570.343750 antigen 3472 1 connect
1240334570.343750 antigen 3472 1 send 2
1240334570.453125 antigen 3472 1 socket
1240334570.546875 antigen 3472 1 connect
1240334570.546875 antigen 3472 1 send 2
1240334570.968750 antigen 3472 1 socket
1240334571.109375 antigen 3472 1 connect
1240334571.109375 antigen 3472 1 send 2
1240334572.15625 antigen 3304 1 send
1240334572.62500 antigen 3472 1 socket
1240334572.234375 antigen 3304 1 ReadFile
1240334572.234375 antigen 3304 1 send
1240334579.281250 antigen 3372 1 send
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyW
```

```
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyW
1240334583.406250 antigen 3248 1 RegOpenKeyW
1240334583.406250 antigen 3248 1 RegQueryValueW
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyW
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyA
1240334583.406250 antigen 3248 1 RegOpenKeyW
1240334583.406250 antigen 3248 1 RegOpenKeyW
1240334583.406250 antigen 3248 1 RegQueryValueW
1240334593.31250 antigen 3472 1 connect
1240334593.390625 antigen 3472 1 socket
1240334593.484375 antigen 3472 1 connect
1240334593.484375 antigen 3472 1 send 2
1240334593.578125 antigen 3472 1 socket
1240334593.687500 antigen 3472 1 connect
1240334593.687500 antigen 3472 1 send 2
1240334593.843750 antigen 3472 1 send 2
1240334594.93750 antigen 3472 1 socket
1240334594.187500 antigen 3472 1 connect
1240334594.187500 antigen 3472 1 send 2
1240334594.281250 antigen 3372 1 send
1240334595.328125 antigen 3472 1 socket
1240334595.421875 antigen 3472 1 connect
1240334595.421875 antigen 3472 1 send 2
1240334595.625000 antigen 3472 1 send 2
1240334595.875000 antigen 3472 1 socket
1240334596.62500 antigen 3472 1 connect
1240334596.62500 antigen 3472 1 send 2
1240334596.234375 antigen 3472 1 socket
1240334596.406250 antigen 3472 1 connect
1240334596.421875 antigen 3472 1 send 2
1240334596.906250 antigen 3472 1 socket
1240334597.109375 antigen 3472 1 connect
1240334597.109375 antigen 3472 1 send 2
1240334597.281250 antigen 3472 1 socket
1240334597.484375 antigen 3472 1 connect
1240334597.484375 antigen 3472 1 send 2
1240334598.0 antigen 3472 1 socket
1240334598.109375 antigen 3472 1 connect
```

```
1240334598.109375 antigen 3472 1 send 2
1240334598.203125 antigen 3472 1 socket
1240334598.312500 antigen 3472 1 connect
1240334598.312500 antigen 3472 1 send 2
1240334598.718750 antigen 3472 1 socket
1240334598.718750 antigen 3472 1 connect
1240334598.718750 antigen 3472 1 socket
1240334598.718750 antigen 3472 1 connect
1240334599.31250 antigen 3472 1 socket
1240334599.125000 antigen 3472 1 connect
1240334599.140625 antigen 3472 1 send 2
1240334599.234375 antigen 3472 1 socket
1240334599.343750 antigen 3472 1 connect
1240334599.343750 antigen 3472 1 send 2
1240334599.406250 antigen 3304 1 GetAsyncKeyState
1240334599.765625 antigen 3472 1 socket
1240334599.765625 antigen 3472 1 connect
1240334599.796875 antigen 3472 1 socket
1240334599.796875 antigen 3472 1 connect
1240334600.109375 antigen 3472 1 socket
1240334600.140625 antigen 3472 1 connect
1240334600.140625 antigen 3472 1 send 2
1240334600.171875 antigen 3472 1 socket
1240334600.203125 antigen 3472 1 connect
1240334600.203125 antigen 3472 1 send 2
1240334600.562500 antigen 3472 1 socket
1240334600.703125 antigen 3472 1 connect
1240334600.703125 antigen 3472 1 send 2
1240334602.46875 antigen 3304 1 ReadFile
1240334602.62500 antigen 3304 1 send
1240334602.640625 antigen 3472 1 socket
1240334602.765625 antigen 3472 1 connect
1240334602.765625 antigen 3472 1 send 2
1240334603.46875 antigen 3472 1 RegQueryValueExW
1240334603.46875 antigen 3472 1 RegQueryValueExW
1240334603.46875 antigen 3472 1 RegQueryValueExW
1240334603.46875 antigen 3472 1 RegQueryValueExW
1240334603.62500 antigen 3472 1 RegQueryValueExW
1240334603.62500 antigen 3472 1 RegQueryValueExW
1240334603.62500 antigen 3472 1 RegQueryValueExW
1240334603.62500 antigen 3472 1 RegQueryValueExW
1240334603.296875 antigen 3472 1 socket
```

```
1240334603.359375 antigen 3472 1 connect
1240334603.359375 antigen 3472 1 send 2
```

D.2 A Sample of Signal Log File

```
1240334965.218750 signal 0 4 0 0 100 0
1240334966.218750 signal 0 4 0 0 100 0
1240334967.218750 signal 0 4 0 0 100 0
1240334968.234375 signal 0 4 0 0 100 0
1240334969.250000 signal 0 4 0 0 100 0
1240334970.250000 signal 0 4 0 0 54.4511 0
1240334971.250000 signal 0 4 0 22.9226 34.0813 0
1240334972.250000 signal 0 4 0 28.6273 0.375 0
1240334973.250000 signal 0 4 0 0 100 0
1240334974.250000 signal 0 4 0 0 100 0
1240334975.250000 signal 0 4 0 22.2789 34.9118 0
1240334976.250000 signal 0 4 30.103 22.2789 0.9375 0
1240334977.250000 signal 0 4 47.7121 24.0824 0.84375 0
1240334978.250000 signal 0 4 0 15.563 0.515625 0
1240334979.250000 signal 0 4 0 19.0849 9.83783 0
1240334980.265625 signal 0 4 0 16.902 0.703125 0
1240334981.281250 signal 0 4 0 20 12.3679 0
1240334982.281250 signal 0 4 0 20.8279 1.24183 0
1240334983.281250 signal 0 4 0 20.8279 12.0982 0
1240334984.281250 signal 0 4 30.103 25.5751 5.01297 0
1240334985.281250 signal 0 4 0 22.9226 0.578125 0
1240334986.281250 signal 0 4 0 9.54243 19.4182 0
1240334987.281250 signal 0 4 30.103 23.5218 7.69253 0
1240334988.281250 signal 0 4 30.103 20.8279 0.703125 0
1240334989.281250 signal 0 4 0 9.54243 19.6244 0
1240334990.281250 signal 0 4 0 20.8279 0.90625 0
1240334991.281250 signal 0 4 0 0 100 0
1240334992.281250 signal 0 4 30.103 24.609 25.176 0
1240334993.281250 signal 0 4 30.103 22.9226 0.453125 0
1240334994.281250 signal 0 4 0 12.0412 0.359375 0
1240334995.281250 signal 0 4 0 20 13.1614 0
1240334996.281250 signal 0 4 0 13.9794 0.390625 0
1240334997.296875 signal 0 4 0 20.8279 17.9295 0
1240334998.296875 signal 0 4 60.206 26.0206 0.984375 0
1240334999.296875 signal 0 4 0 18.0618 0.96875 0
1240335000.296875 signal 0 4 0 20.8279 0.78125 0
```

```
1240335001.296875 signal 0 4 0 0 100 0
1240335002.296875 signal 0 4 0 15.563 23.8032 0
1240335003.296875 signal 0 4 0 18.0618 0.40625 0
1240335004.296875 signal 0 4 0 16.902 10.4208 0
1240335005.296875 signal 0 4 0 22.9226 0.834174 0
1240335006.296875 signal 0 4 0 19.0849 0.828125 0
1240335007.296875 signal 0 4 0 12.0412 0.4375 0
1240335008.296875 signal 0 4 0 15.563 15.8869 0
1240335009.296875 signal 0 4 47.7121 25.1055 0.420295 0
1240335010.296875 signal 0 4 0 21.5836 0.546875 0
1240335011.296875 signal 0 4 0 15.563 7.69253 0
1240335012.296875 signal 0 4 0 16.902 0.390625 0
1240335013.296875 signal 0 4 0 22.9226 14.1845 0
1240335014.296875 signal 0 4 0 20 0.75 0
1240335015.296875 signal 0 4 0 0 0.40625 0
1240335016.296875 signal 0 4 0 21.5836 11.8257 0
1240335017.296875 signal 0 4 0 16.902 7.04706 0
1240335018.296875 signal 0 4 30.103 24.0824 2.42925 0
1240335019.296875 signal 0 4 0 6.0206 100 0
1240335020.296875 signal 0 4 0 12.0412 18.3633 0
1240335021.296875 signal 0 4 0 21.5836 1.64344 0
1240335022.296875 signal 0 4 0 21.5836 0.84375 0
1240335023.296875 signal 0 4 0 0 100 0
1240335024.296875 signal 0 4 0 6.0206 100 0
1240335025.296875 signal 0 4 0 13.9794 23.2699 0
1240335026.296875 signal 0 4 0 15.563 9.24201 0
1240335027.296875 signal 0 4 0 24.609 0.4375 0
1240335028.296875 signal 0 4 0 18.0618 15.1703 0
1240335029.296875 signal 0 4 0 19.0849 0.75 0
1240335030.296875 signal 0 4 0 20.8279 3.19292 0
1240335031.296875 signal 0 4 0 15.563 13.4208 0
1240335032.296875 signal 0 4 0 0 100 0
1240335033.296875 signal 0 4 0 0 100 0
1240335034.296875 signal 0 4 0 0 100 0
1240335035.296875 signal 0 4 0 16.902 33.8392 0
1240335036.296875 signal 0 4 0 19.0849 0.40625 0
1240335037.296875 signal 0 4 0 9.54243 100 0
1240335038.296875 signal 0 4 0 0 100 0
1240335039.296875 signal 0 4 0 0 100 0
1240335040.296875 signal 0 4 0 0 100 0
1240335041.296875 signal 0 4 0 0 100 0
1240335042.296875 signal 0 4 0 0 100 0
```

```
1240335043.296875 signal 0 4 0 0 100 0
1240335044.296875 signal 0 4 0 0 100 0
1240335045.296875 signal 0 4 0 9.54243 59.0887 0
1240335046.296875 signal 0 4 0 0 100 0
1240335047.296875 signal 0 4 0 0 100 0
1240335048.296875 signal 0 4 0 0 100 0
1240335049.296875 signal 0 4 0 0 100 0
1240335050.296875 signal 0 4 0 0 100 0
1240335051.296875 signal 0 4 0 0 100 0
1240335052.296875 signal 0 4 0 9.54243 100 0
1240335053.296875 signal 0 4 0 13.9794 63.5398 0
1240335054.296875 signal 0 4 0 6.0206 100 0
1240335055.296875 signal 0 4 0 0 100 0
1240335056.296875 signal 0 4 0 12.0412 28.628 0
1240335057.296875 signal 0 4 0 0 100 0
1240335058.296875 signal 0 4 0 18.0618 16.3543 0
1240335059.296875 signal 0 4 0 9.54243 7.04706 0
1240335060.296875 signal 0 4 30.103 25.1055 0.5625 0
1240335061.296875 signal 0 4 0 12.0412 0.5 0
1240335062.296875 signal 0 4 0 16.902 10.9916 0
1240335063.296875 signal 0 4 0 12.0412 0.46875 0
1240335064.296875 signal 0 4 0 18.0618 16.585 0
1240335065.296875 signal 0 4 0 9.54243 100 0
1240335066.296875 signal 0 4 0 15.563 11.8257 0
1240335067.296875 signal 0 4 0 13.9794 0.84375 0
1240335068.296875 signal 0 4 0 0 100 0
1240335069.296875 signal 0 4 0 18.0618 26.3228 0
1240335070.296875 signal 0 4 30.103 22.2789 0.734375 0
1240335071.296875 signal 0 4 0 13.9794 7.04706 0
1240335072.296875 signal 0 4 30.103 20.8279 0.671875 0
1240335073.296875 signal 0 4 0 15.563 0.46875 0
1240335074.296875 signal 0 4 0 20 25.6734 0
1240335075.296875 signal 0 4 0 0 0.578125 0
1240335076.296875 signal 0 4 0 20 13.4208 0
1240335077.328125 signal 0 4 0 0 100 0
1240335078.328125 signal 0 4 0 16.902 22.9084 0
1240335079.328125 signal 0 4 30.103 9.54243 100 0
1240335080.328125 signal 0 4 0 15.563 10.9916 0
1240335081.328125 signal 0 4 0 16.902 0.546875 0
1240335082.328125 signal 0 4 0 19.0849 17.2656 0
1240335083.328125 signal 0 4 0 9.54243 100 0
1240335084.328125 signal 0 4 0 0 100 0
```

```
1240335085.328125 signal 0 4 0 0 100 0
1240335086.328125 signal 0 4 0 15.563 37.6861 0
1240335087.328125 signal 0 4 0 22.2789 0.65625 0
1240335088.328125 signal 0 4 0 40.3407 30.202 0
1240335089.328125 signal 0 4 0 41.7981 0.78125 0
1240335090.328125 signal 0 4 0 42.6708 9.24201 0
1240335091.328125 signal 0 4 0 41.5836 3.19292 0
1240335092.328125 signal 0 4 0 31.364 0.546875 0
1240335093.328125 signal 0 4 0 21.5836 0.75 0
1240335094.328125 signal 0 4 0 9.54243 7.04706 0
1240335095.328125 signal 0 4 0 20 9.83783 0
1240335096.328125 signal 0 4 47.7121 22.2789 13.9323 0
1240335097.328125 signal 0 4 0 15.563 0.296875 0
1240335098.328125 signal 0 4 0 18.0618 13.6778 0
1240335099.328125 signal 0 4 30.103 22.2789 0.578125 0
1240335100.328125 signal 0 4 0 13.9794 0.46875 0
1240335101.328125 signal 0 4 0 18.0618 17.71 0
1240335102.328125 signal 0 4 0 9.54243 100 0
1240335103.328125 signal 0 4 0 22.2789 27.1132 0
1240335104.328125 signal 0 4 0 18.0618 0.53125 0
1240335105.328125 signal 0 4 0 18.0618 2.42925 0
1240335106.328125 signal 0 4 0 20.8279 0.5 0
1240335107.328125 signal 0 4 0 20.8279 14.6819 0
1240335108.328125 signal 0 4 0 0 100 0
1240335109.343750 signal 0 4 30.103 25.1055 19.2105 0
1240335110.343750 signal 0 4 0 13.9794 0.578125 0
1240335111.343750 signal 0 4 0 19.0849 9.83783 0
1240335112.343750 signal 0 4 0 12.0412 0.234375 0
1240335113.343750 signal 0 4 0 0 100 0
1240335114.343750 signal 0 4 0 19.0849 34.3212 0
1240335115.343750 signal 0 4 0 6.0206 100 0
1240335116.343750 signal 0 4 0 18.0618 8.32299 0
1240335117.343750 signal 0 4 47.7121 22.9226 13.4208 0
1240335118.343750 signal 0 4 30.103 20 16.1216 0
1240335119.343750 signal 0 4 0 0 100 0
1240335120.343750 signal 0 4 0 25.5751 20.8294 0
1240335121.343750 signal 0 4 0 9.54243 0.390625 0
1240335122.343750 signal 0 4 47.7121 25.5751 0.953125 0
1240335123.343750 signal 0 4 30.103 20 0.578125 0
1240335124.343750 signal 0 4 0 9.54243 8.00959 0
1240335125.343750 signal 0 4 0 16.902 0.578125 0
1240335126.343750 signal 0 4 0 15.563 3.19292 0
```

```
1240335127.343750 signal 0 4 0 13.9794 0.53125 0
1240335128.343750 signal 0 4 0 15.563 8.32299 0
1240335129.343750 signal 0 4 30.103 20 2.81378 0
1240335130.390625 signal 0 4 0 13.9794 0.625 0
1240335131.390625 signal 0 4 0 18.0618 14.6819 0
1240335132.390625 signal 0 4 0 16.902 0.578125 0
1240335133.390625 signal 0 4 47.7121 26.0206 11.5505 0
1240335134.390625 signal 0 4 0 19.0849 0 0
1240335135.390625 signal 0 4 47.7121 25.1055 13.1614 0
1240335136.390625 signal 0 4 0 13.9794 0.453125 0
1240335137.390625 signal 0 4 0 15.563 6.04909 0
1240335138.390625 signal 0 4 0 22.2789 8.63281 0
1240335139.390625 signal 0 4 60.206 25.1055 9.83783 0
1240335140.390625 signal 0 4 30.103 22.2789 0.420295 0
1240335141.390625 signal 0 4 0 13.9794 0.84375 0
1240335142.390625 signal 0 4 0 18.0618 0.984375 0
1240335143.390625 signal 0 4 0 9.54243 100 0
1240335144.390625 signal 0 4 0 9.54243 15.6502 0
1240335145.390625 signal 0 4 0 22.9226 0 0
1240335146.390625 signal 0 4 0 9.54243 8.93912 0
1240335147.390625 signal 0 4 30.103 23.5218 0.59375 0
1240335148.390625 signal 0 4 0 13.9794 0.5625 0
1240335149.390625 signal 0 4 0 6.0206 17.4887 0
1240335150.390625 signal 0 4 0 0 100 0
1240335151.390625 signal 0 4 0 9.54243 16.3543 0
1240335152.390625 signal 0 4 0 18.0618 0.96875 0
1240335153.390625 signal 0 4 0 0 0.65625 0
1240335154.390625 signal 0 4 0 15.563 13.6778 0
1240335155.390625 signal 0 4 30.103 22.9226 0.90625 0
1240335156.390625 signal 0 4 0 9.54243 100 0
1240335157.390625 signal 0 4 0 18.0618 8.93912 0
1240335158.390625 signal 0 4 0 23.5218 9.83783 0
1240335159.390625 signal 0 4 0 9.54243 100 0
1240335160.406250 signal 0 4 47.7121 36.2583 32.8489 0
1240335161.531250 signal 0 4 0 36.1236 1.64344 0
1240335162.531250 signal 0 4 0 9.54243 2.81378 0
1240335163.531250 signal 0 4 0 18.0618 8.32299 0
1240335164.531250 signal 0 4 0 9.54243 0.578125 0
1240335165.531250 signal 0 4 0 0 100 0
1240335166.531250 signal 0 4 0 20.8279 6.71847 0
1240335167.531250 signal 0 4 0 13.9794 15.1703 0
1240335168.531250 signal 0 4 0 9.54243 100 0
```

```
1240335169.531250 signal 0 4 0 0 100 0
1240335170.531250 signal 0 4 0 0 100 0
1240335171.531250 signal 0 4 0 19.0849 33.4721 0
1240335172.531250 signal 0 4 0 13.9794 0.84375 0
1240335173.531250 signal 0 4 0 0 100 0
1240335174.531250 signal 0 4 0 0 100 0
1240335175.531250 signal 0 4 0 9.54243 35.9435 0
1240335176.531250 signal 0 4 0 19.0849 0.6875 0
1240335177.531250 signal 0 4 0 22.2789 5.7081 0
1240335178.531250 signal 0 4 0 0 100 0
1240335179.531250 signal 0 4 0 22.9226 18.7902 0
1240335180.531250 signal 0 4 0 9.54243 2.81378 0
1240335181.531250 signal 0 4 30.103 23.5218 0.834174 0
1240335182.531250 signal 0 4 0 9.54243 100 0
1240335183.531250 signal 0 4 0 0 100 0
1240335184.531250 signal 0 4 0 20.8279 33.4721 0
1240335185.531250 signal 0 4 0 13.9794 0.421875 0
1240335186.531250 signal 0 4 47.7121 22.9226 0 0
1240335187.531250 signal 0 4 0 18.0618 9.54156 0
1240335188.531250 signal 0 4 0 0 100 0
1240335189.531250 signal 0 4 0 0 100 0
1240335190.531250 signal 0 4 0 0 100 0
1240335191.531250 signal 0 4 0 0 100 0
1240335192.531250 signal 0 4 0 0 100 0
1240335193.531250 signal 0 4 0 0 100 0
1240335194.531250 signal 0 4 0 0 51.0656 0
1240335195.531250 signal 0 4 0 0 100 0
1240335196.531250 signal 0 4 0 0 100 0
1240335197.531250 signal 0 4 0 0 100 0
1240335198.531250 signal 0 4 0 0 100 0
1240335199.531250 signal 0 4 0 0 100 0
1240335200.531250 signal 0 4 0 0 100 0
1240335201.531250 signal 0 4 0 0 100 0
1240335202.531250 signal 0 4 0 0 100 0
1240335203.531250 signal 0 4 0 0 100 0
```