Multi-Agent Malicious Behaviour Detection

by

Ryan P.D. Wegner

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Doctor of Philosophy

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
July 2012

Thesis advisor | Author

John E. Anderson | **Ryan P.D. Wegner**

Multi-Agent Malicious Behaviour Detection

Abstract

This research presents a novel technique termed *Multi-Agent Malicious Behaviour Detection*. The goal of Multi-Agent Malicious Behaviour Detection is to provide infrastructure to allow for the detection and observation of *malicious multi-agent systems* in computer network environments. This research explores combinations of machine learning techniques and fuses them with a multi-agent approach to malicious behaviour detection that effectively blends human expertise from network defenders with modern artificial intelligence. Detection in this approach focuses on identifying multiple distributed *malicious software agents* cooperating to achieve a malicious goal in a complex dynamic environment. A significant portion of this approach involves developing *Multi-Agent Malicious Behaviour Detection Agents* capable of supporting interaction with malicious multi-agent systems, while providing network defenders a mechanism for improving detection capability through interaction with the Multi-Agent Malicious Behaviour Detection system. Success of the approach depends on the Multi-Agent Malicious Behaviour Detection system's capability to adapt to evolving malicious multi-agent system communications, even as the malicious software agents in network environments vary in their degree of autonomy and intelligence. The Multi-Agent Malicious Behaviour Detection system aims to take advantage of detectable behaviours that individual malicious software agents as well as malicious multi-agent systems are likely to exhibit, including: beaconing, denying, propagating, ex-filtrating, updating and mimicking. This thesis research involves the design of this framework, its implementation into a working tool, and its evaluation using network data generated by an enterprise class network appliance to simulate both a standard educational network and an educational network containing malware traffic.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis was created by the combined efforts of several people who have worked with me, inspired me, and helped me when the writing got rough. I would like to thank the following people. My advisor Dr. John Anderson. My committee members, Dr. David Scuse, Dr. David Whyte, and Dr. Robert Macleod. Mr. Brian Ritchot for giving me the opportunity to go back and finish what I started years ago. Finally, my family and friends that I have neglected while working on this.

I dedicate this thesis to the memory of my parents Robert and Cherie Wegner. To my wife for supporting me and to my three amazing children (Brooke, Ethan, and Kaylah) for understanding. Daddy doesn't have to work all the time anymore.

Chapter 1

Introduction

1.1 Overview

In 2010 Google announced that its offshore offices were the victims of a "highly sophisticated and targeted attack" on their corporate infrastructure [Deibert and Rohozinskli, 2010]. Termed *Aurora*, the attack manifested as infestations of malicious software that allowed attackers to steal intellectual property from over 30 other companies that Google later identified [Stamos, 2010]. The attacks typically involved social engineering that led to the download and installation of malicious software enabling the attacker to establish a presence on a company's internal network. Once inside, the attackers could elevate their privileges, exploit virtual private networks (VPNs), and steal intellectual property.

The attacks on Google are an example of the type of malicious multi-agent system that this research is concerned with detecting: cooperative goal-oriented multi-agent systems with malicious intent. The technique that attackers employed against Google is similar across a number of reported infections [Deibert and Rohozinskli, 2009, 2010]: the distribution of malicious software agents across multiple victims on diverse networks enabling a malicious multi-agent system to exploit vulnerable companies stealing intellectual property. Like any other software agent, each malicious software agent involved in Aurora attacks had a task to complete, and being distributed across a network required some amount of autonomy to complete those tasks. It's not clear what level of autonomy each instance required to complete its goals - perhaps requiring interactions from operators in a teleautonomous fashion. However, at the very least the malicious software agents were able to signal other agents in the malicious multi-agent system, collect information and relay that information out to controlling entities in the malicious multi-agent system. I define the process of modelling the behaviours exhibited by malicious multi-agent systems, such as the one responsible for the attack on Google, *Multi-Agent Malicious Behaviour Detection*. The purpose of this thesis is the development of a framework for Multi-Agent Malicious Behaviour Detection, to aid network defenders (human experts responsible for the protection of computer networks) in the detection, mitigation, and study of malicious multi-agent systems. This thesis research involves the design of this framework and its implementation into a working tool. The framework and system are evaluated using network data generated by an enterprise class network appliance to simulate both a standard educational network and an educational network containing malware traffic.

In particular, my framework supports the following activities:

- reads network traffic in real-time, extracting features of interest,
- supports the learning of malicious multi-agent system behaviours based on hu-

man knowledge,

- encourages human-machine interaction to discover novel malicious multi-agent system behaviour,
- engages in malicious agent communication, thereby revealing more about their behaviour,
- integrates external detection system output to enhance system knowledge,
- and evaluates machine learning techniques for malicious multi-agent system detection.

The remainder of this chapter describes the evolution of malicious software agents, and outlines some of the malicious software developer's motivations for continuing to improve malicious software agents and deploy malicious multi-agent systems. Next, I discuss the challenges associated with work detecting malicious multi-agent systems, and motivate the need for Multi-Agent Malicious Behaviour Detection. Finally, I present a hypothesis and research questions.

1.2 Malicious Software Agent Evolution

Malicious software agents have evolved drastically over the past 40 years as the speed of technological advances continues to outpace our ability to secure those same technologies from exploitation. Consider recent advances in pacemaker and insulin pump technologies, which use wireless signals to control the device and may in the future allow medical practitioners to monitor and adjust settings on the device through wireless communication alone. The benefits of such technological advances are clearly important. However, current network security techniques are not truly prepared for the challenge of securing such a wireless device from being exploited or tampered with [Maisel and Tadayoshi, 2010]. For example, recently an attack was devised that allows an attacker to deliver fatal doses of insulin to diabetic patients Goodin [2011]. The potential for a malicious multi-agent system consisting of malicious software agents exploiting pacemakers or insulin pumps may sound like science fiction. However, history has shown that new technologies are often exploited early in their deployment. The following comprise a brief historical view of some of the major events that demonstrate the speed with which malware has progressed to date. This progress in malware, leading to the existence of sophisticated malicious multi-agent systems, is a primary motivator for Multi-Agent Malicious Behaviour Detection.

1.2.1 Early Malicious Software Agents

The Creeper virus is one of the earliest known examples of a primitive malicious software agent that demonstrated the ability to copy itself from an infected host system to other remote systems via the Advanced Research Projects Agency Network (ARPANET) in 1971 [Chen and Robert, 2005]. Other malicious software agents followed, many of which shared the self-replicating characteristic which originated the term *computer virus*. Examples include: ANIMAL [Dewdney, 1988], intended to be non-malicious malware that copied itself throughout file systems on multi-user UNIVACs machines and copied onto tapes shared to other machines; Elk Cloner [Spafford, 1990], a virus capable of infecting the boot sector of Apple II machines in 1982; and

Figure 1.1: Machines vulnerable to primitive malicious software agents. Left: IBM Mainframe. Top: UNIVAC. Bottom: Apple II. Right: DEC VAX.

Christmas Tree EXEC [Chen, 2003], responsible for disrupting mainframe operation in 1987.

One well known malicious software agent in computer security circles is the Morris Worm [Chen, 2003]. In computer security, a worm refers to a self replicating malware computer program that spreads from a host machine to other machines. In 1988 the Morris Worm was capable of infecting DEC VAX machines running 4BSD as well as Sun 3 systems. The Morris Worm was also the first worm written that resulted in the author being convicted under the 1986 Computer Fraud and Abuse Act. Some estimates claim it resulted in 10% of machines connected to ARPNET rendered useless. Once installed the worm replicates on a host machine consuming all available resources [Spafford, 1988]. Figure 1.1 illustrates some early machines that have all been victims of early malicious software agents. Even early forms of malware share some characteristics of malicious multi-agent systems. From early on the motivation for self-replication has influenced malicious software agent development. While arguably the Creeper virus was not in any way aware of other virus instances, it shares the same distributed characteristic as modern malicious multi-agent systems.

1.2.2 Trends in Malicious Multi-Agent Systems

The popularity of the Internet continues to grow and is gradually becoming ubiquitous within Canadian society. According to Statistics Canada 73% of Canadians aged 16 or over accessed the Internet for personal reasons in 2007, up from 68% in 2005 [Canada, 2009]. Increasing Internet usage has enlarged the attack surface for malicious software agents. In the early 1970's damage caused by malicious software agents was limited by the types of networks that existed and their relative specialty. In the last 15 years there have been several instances of malicious multi-agent systems causing significant financial losses to a number of diverse unrelated institutions. Consider the following two examples of malicious multi-agent systems causing significant financial losses.

The ILOVEYOU worm, an example of a malicious software agent which appeared in May 2000, was delivered as an email attachment to several in-boxes around the world [Chen and Robert, 2005]. The worm propagated by attaching a malicious visual basic script to email destined to users who's email addresses resided in the originating hosts email address book. The rule of thumb at the time was to ignore attachments from unfamiliar email addresses. The worm deceived recipients by using the infected hosts contact list to choose targets. The familiar email addresses misled many users to open the attachment. The attachment also appeared harmless, it was named LOVE-LETTER-FOR-YOU.TXT.vbs with a subject line of ILOVEYOU. The attachment name would appear to be a text file, since a feature of many flavors of Microsoft Windows hides the file extension, .vbs. The worm propagated so successfully that it eventually infected over 50 million hosts by 13 May 2000. It also caused an estimated $7.0 billion in damages [Chen and Robert, 2005].

Conficker is a widespread worm that is estimated to have infected over 7 million machines worldwide in 2009 and is one of the most complex examples of malicious multi-agent systems to date. Conficker is an example of an evolving malicious multi-agent system actively maintained by a group of unknown authors who continually update the worm improving its spreading efficiency and incorporating defence mechanisms to ensure its continued propagation. There are a total five variations of Conficker, named A, B, C, D and E [Leder and Tilmann, 2009]. The major cost associated with Conficker is securing networks from its propagation and removing existing infestations of the the worm. Other economic effects of the infestation include the costs related to spam generated by variation E of the worm, significant increases in network congestion and the impact of scare-ware installed in variation E [Porras et al., 2009].

Whereas malicious software agents five to ten years ago focused on mischief and disruption of services, more recent activity, including large scale malicious multi-agent systems, often referred to as botnets, seem to be purposed for financial gain [Wang et al., 2010; Bacher et al., 2008; Rieck et al., 2010]. The next section addresses various motivations for the evolution of malicious multi-agent systems.

1.3 Motivations

There are several motivations for malware designers to continue to create more complex malicious multi-agent systems. The motivations range from profit, to mischief, to social activism.

1.3.1 Criminal Enterprise

The Internet hosts a multi-billion dollar consumer environment. In 2006 Statistics Canada [Canada, 2009] released figures indicating that Internet sales totalled $62.7 billion, a figure that has been rising drastically since 2000. Online transactions have been targeted by criminal enterprise for some time now [Franklin and Paxson, 2007]. The actual mechanics of online fraud have included web site phishing [Lauinger et al., 2010], username and password harvesting [Rieck et al., 2010; Stone-Gross et al., 2009], credit card stealing [Stone-Gross et al., 2009], and email spamming [Lauinger et al., 2010].

One interesting example of criminal activities highlighted by Peng et al. [2009] is the prevalence of click bots. Click bots are malicious software agents that run on the victim machine with the sole purpose of clicking online advertisements to either increase revenue for syndicators or force competing advertisers to pay high advertising fees for fake advertisement clicks, since some online advertisement structures include pay per click payments. Criminals can take advantage of malicious multi-agent systems to deploy large numbers of click bots as malicious software agents to influence revenues for online advertisers, publishers and syndicators. More sophisticated click bots are likely to remain active longer, increasing the opportunity for profit.

1.3.2 Espionage

Online espionage is difficult to quantify. However, its existence has been documented in some cases [Deibert and Rohozinskli, 2009, 2010]. In particular corporate espionage involves hiring a company or contracting an individual to exploit technology in order to gain access to information that will give one company a competitive edge against competitors. This could include information about upcoming products, marketing strategies, planned acquisitions, etc. In private industry, if one company can learn the closely held secrets of another company it can profit by exploiting that information.

Espionage relies on malicious software agents capable of mimicking common network traffic to avoid detection and ex-filtrating data to exploit competitors (Section 1.5.4).

1.3.3 Social Activists

The Internet acts as a mass media distribution network. As such, any group looking to spread its point of view whether it be religious, political, philosophical or malicious can reach millions across the world using the Internet. Web forums, blogs and social networking sites are all examples of technologies that allow people to share their ideas. The malicious activist exploits such technologies in an attempt to spread their message to people who are not actively looking for it, or to suppress the information of rival groups [McCaughey and Ayers, 2003]. For example, a group of environmental activists might try to deface an oil company's web page, portraying the oil company in a bad light. Equally, political parties may try to force a rival party's web pages offline through a number of techniques.

Malicious multi-agent systems are an effective tool for social activists. As malicious multi-agent systems propagate, they enlist more and more malicious software agents that can potentially participate in a distributed denial of service (DDoS). Propagate and deny (Section 1.5.4) are both key to achieving the goals of the social activist.

1.3.4 Mischief

Finally, there are those individuals or groups who enjoy exploiting technology for the challenge of it and in general either want to cause mischief, or accept that a side effect of their tinkering involves mischief [Alhabeeb et al., 2010].

1.4 Malicious Multi-Agent Systems

Malicious software agents continue to evolve today, making identifying and tracking malicious multi-agent systems a challenging problem. Consider an intelligent malicious software agent operating in the context of a malicious multi-agent system. Assuming that the malicious software agent is designed to be both subtle and covert, it will surely disguise itself by mimicking one of several protocols commonly seen on a target network. For example, on observing SETI@home traffic on a network, a malicious software agent might choose to disguise its communications to look like SETI@home packets. Even if the malicious software agent is not intelligent enough to both recognize SETI@home traffic and disguise itself as that, a malicious multi-agent system (perhaps including human operators) could use malicious software agents to snoop a network and, recognizing the SETI@home traffic, modify their communications to appear similar. It would then be difficult for a network defender to distinguish between the malicious software agent's mimicked SETI@home and the legitimate SETI@home. SETI@home is just one example of a protocol that can hide malicious multi-agent system communications. DNS, HTTP, email attachments, all of which can be potential containers for malicious multi-agent system communications.

Further, suppose a network defender knows that some malicious multi-agent system communications exist in SETI@home traffic. There are limited options available for preventing the individual malicious software agent communications from leaking information out of the network. First, the network defender might decide that SETI@home traffic should be blocked entirely from the protected network. An intelligent malicious multi-agent system would likely recognize a drop in perceived SETI@home traffic from some of the malicious software agents and adjust the malicious multi-agent system's communications to some other common protocol on the target network. Effectively the network defender might suppress a few of the malicious software agents from communicating for a short time, but risks upsetting the network clients. There are a finite number of legitimate protocols a network defender can block before impacting the clients to a degree that they no longer find the host network valuable. Second, the network defender might try to distinguish between malicious SETI@home and benign SETI@home. Such a task requires significant manual effort. The network defender must learn how to recognize the SETI@home protocol and know enough about the protocol to understand what information the packets contain. Then the network defender must be able to identify features in the protocol that differentiate between the benign SETI@home and the malicious multi-agent system communications mimicking SETI@home. Third, the network defender could wait until there is an update or patch available for their systems that will recognize the malicious software agents without the intense manual labour required on the part of the network defender. Assuming an update or patch is made available, there is still the time between the infection and the update/patch where the malicious software agent is able to leak information from or cause damage to the target network, and being part of a larger malicious multi-agent system, the malicious multi-agent system could recognize the loss of communications from individual malicious software agents and adapt the malicious software agents that are still successfully communicating.

The more difficult scenario arises when the network defender does not know the network has been infected. Suppose a malicious multi-agent system has established itself on a target network, with several malicious software agents distributed across a number of machines. On instruction from controlling agents in the malicious multi-agent system the malicious software agents have disguised themselves by mimicking one of tens or hundreds of different protocols in the network and are leaking information from the network to the greater malicious multi-agent system. The network defender can monitor the network with a variety of commercially available security products, but they are reactive in nature. In order to find malware, the protocols have to have been previously identified and distinguishable from the benign variations of common network protocols. The network defender is tasked with identifying malicious multi-agent system communications while there is no certainty that any malicious software agents are present. The problem is comparable to searching for a needle in a haystack, when it is not clear that there is a needle to be found.

Detecting malicious multi-agent system communications is somewhat analogous to medical diagnostics [Zelonis, 2004]. At any point in time the human body is assaulted by a number of different potential viral infections, miniature biological multi-agent systems. These miniature biological multi-agent systems aren't recognized until symptoms are present. When symptoms are present, a medical professional may be tasked to diagnose the causes of the various symptoms. In some cases an infection is identified, or at least a set of likely causes are identified and treated. However, the most insidious infections are those that do not present early symptoms or present symptoms that do not distinguish the infection from a variety of other common infections. For example, consider a deadly biological virus that masks itself as a common cold. The impact of such a virus would be dramatic, and threatening to the human host, as it would evade standard diagnosis techniques.

Just as diseases may interact and present different symptoms than they would separately, the act of diagnosis in a multi-agent software setting is similarly complicated. However, an important distinguishing factor between medical diagnosis and Multi-Agent Malicious Behaviour Detection is that in the case of intelligent covert malicious software agents, the impact might not present itself in the network in the same way a biological virus presents in a human host. If, for example, a malicious multi-agent system is tasked with stealing banking information, the real world impact might be identified when a victim's credit card statement reveals fraudulent charges. Tying those charges back to an individual malicious software agent or a larger malicious multi-agent system on an online market place would be difficult and there is no incentive for the malicious multi-agent system to overtly damage the target network as long as the malicious software agents are recovering valuable information.

1.5 Detecting Malicious Multi-Agent Systems

As previously discussed, malicious multi-agent systems are evolving, and detecting them is difficult. However, various areas of artificial intelligence are advancing faster than ever before, as technology in general continues to improve exponentially. Techniques that were impossible, due to technical limitations, are now becoming reasonable and these advancements require that researchers in areas such as computer security constantly revisit domains such as artificial intelligence for potential problem solutions.

1.5.1 Machine Learning

Ensuring that the machine learning and computer security keep pace with each other is important in advancing both disciplines, and necessary if network defenders are to keep up with evolving malicious multi-agent systems. Each year machine learning algorithms are enhanced and validated against a variety of problem sets, such as DNA classification and object recognition in computer vision [Leistner et al., 2010; Jensen and Bateman, 2011; Kamath et al., 2011]. While useful in these particular problems, the application of such techniques to new and distinct problem areas is a way to further validate these techniques, and advance machine learning by noting where the techniques can be improved. In my thesis research, the novelty of using machine learning to detect, discover, and mitigate malicious multi-agent systems is realized by incorporating recently published machine learning techniques into a Multi-Agent Malicious Behaviour Detection system that supports network defenders in detecting modern malicious multi-agent systems. It is also likely that good potential online machine learning algorithms would have previously been discounted for use in security applications because of the speed of networks relative to resources available to these algorithms. However, as processors and memory scale up, these previous techniques may become more suitable. Chapter 6 compares a number of these approaches, and illustrates several are suitable choices for this framework.

One of the difficult and novel aspects of this research is building a Multi-Agent Malicious Behaviour Detection framework that can leverage modern machine learning techniques and apply them to the problem of malicious multi-agent system detection. Constructing such a framework is difficult, as it requires the capability to frame malicious multi-agent system detection into a context where machine learning is applicable. While this work is not intended to make substantial contributions to machine learning directly, it is expected that each problem that shows a significant gain from using the latest machine learning techniques has the effect of motivating further research in machine learning.

In addition to treating groups of malicious software agent as malicious multi-agent system, the framework I present is itself a Multi-Agent approach. The following section elaborates on the multi-agent perspective as a solution.

1.5.2 Multi-Agent Approach

Multi-Agent Malicious Behaviour Detection is a multi-agent approach to detecting, discovering, and mitigating malicious multi-agent systems. Increases in the number of cores on modern multi-core processors have made multi-agent approaches more feasible in the computer security domain. Multi-agent systems have been deployed for malware detection before, and I will present a number of them in Chapter 3, in order to contrast my approach. Increasingly, multi-agent approaches are more effective as resources on machines are becoming easier to share between multiple agents. The novelty here is the division of effort among the Multi-Agent Malicious Behaviour Detection Agents, and the usage of techniques that were previously not readily available. For example, I take advantage of advances in the RabbitMQ AMQP system for reliable communication between Multi-Agent Malicious Behaviour Detection Agents, further discussed in Section 5.4.1. Additionally, deep packet inspection has been made available to high level programming languages through projects such as Sharp Pcap. There are many disparate components that make up the Multi-Agent Malicious Behaviour Detection framework presented as part of this research, and the truly difficult and novel aspect of this work is bringing them all together to confront the advances in malicious multi-agent systems. As discussed in Section 1.4, intelligent malware is making the problem of detecting malicious multi-agent systems ever more difficult. Here the goal is to ensure that a framework exists enabling multi-agent systems to evolve to cope with the challenge that malicious multi-agent systems pose.

1.5.3 Human-Machine Interaction

The holy grail of computer security is a fully autonomous system capable of keeping pace with evolving malicious multi-agent systems and ensuring a near perfect detection and mitigation capability. However, as I will discuss in the next two chapters, the state of the art in computer security is still far from that goal. Much of computer security is still a manual process. This work provides novel insight into the division of labour between autonomous systems and human network defenders. As part of my framework, network defenders will describe potentially malicious multi-agent system behaviour using a human-readable format. The autonomous parts of the Multi-Agent Malicious Behaviour Detection system derive hypotheses from complex network features, which are difficult for network defenders to understand. The hypotheses regarding malicious multi-agent system communications are then grounded to the human-readable knowledge to improve human-machine interactions. Tying the Multi-Agent Malicious Behaviour Detection system's representation of traffic to something a network defender can understand is important for improving human machine interaction.

Given that the human network defender and autonomous parts of the Multi-Agent Malicious Behaviour Detection system do not share identical representations of traffic, the system will make hypothesis about traffic that do not precisely match what the human network defender has tasked the system to look for. The system will provide some measure of confidence, and the human network defender can use that confidence measure to decide whether a hypothesis is worth following up on. The ability to possibly exceed the specifications of humans by looking for behaviour that looks *something like* what it has been tasked to search for means that the system can possibly discover novel malicious multi-agent behaviour.

1.5.4 Malicious Multi-Agent Behaviours

To accomplish its goals, any malicious multi-agent system will have to engage in a collection of activities that have observable effects across a network. Multi-Agent Malicious Behaviour Detection relies on both being able to detect and participate in malicious multi-agent system behaviours. Those behaviours will be discussed throughout this research, and are described here.

Beacon A mechanism used by malicious software agents to announce their presence to some controlling infrastructure in the malicious multi-agent system. Beacons typically have some periodicity, but may be sent out at random intervals to decrease the likelihood of being detected. A repeated beacon increases the likelihood that the malicious multi-agent system can maintain contact with individual malicious software agents. While beacons are simple in concept, they provide the Multi-Agent Malicious Behaviour Detection system with something to detect, as they are pervasive in malicious multi-agent systems [Dash et al., 2006].

Deny Denial of service has long been the result of many instances of malware. Examples include the Morris Worm [Spafford, 1988], the ILOVEYOU Worm [Chen and Robert, 2005] and Conficker [Porras et al., 2009]. There are numerous examples where self propagating worms have rendered mail servers useless or congested network traffic to a degree that legitimate traffic can no longer reach its destination. Some viruses and worms have been known to simply attempt replication within the infected host tying up system resources.

Trojans and viruses have been known to selectively shut down services on the host machine. For example, there have been cases where a trojan was capable of shutting off the host anti-virus software. Some variations of Conficker (more precisely a worm) are able to disrupt the host's ability to use domain name resolution services [Leder and Tilmann, 2009].

Web hijacking and phishing sites often prevent the victim from visiting targeted websites or direct the user to their own websites. In essence they deny the victim the services that would have been provided by the website the victim intended to visit [Wurzinger et al., 2009].

With respect to denial of service, Multi-Agent Malicious Behaviour Detection looks for a specific type of denial of service, the distributed denial of service attack (DDoS). This form of denial of service involves multiple malicious software agents making connection requests, or some other service request, to the same server rapidly and repeatedly. The target server becomes overwhelmed with fake requests from the malicious multi-agent system and is unable to serve legitimate requests. There have been many instances of this occurring throughout the last decade (see [Lawniczak et al., 2009]).

When malicious software agents deny service to a target host they produce a significant amount of traffic, that is detectable. The distributed nature of the attack can aid in tracking the malicious multi-agent system.

Propagate To be successful, a malicious multi-agent system must propagate to additional hosts, distributing malicious software agents throughout the target network. A malicious software agent may contain a mechanism to infect other hosts on the local network or embed itself into communications between the infected host and other hosts, as is typical with traditional malware [Chen and Robert, 2005].

The malicious multi-agent system propagation can exhibit itself by a sharp increase in email traffic or duplicate data being sent frequently within a group of machines. The importance of propagation is that in order to propagate, data must be passed from one machine to another, therefore the physical event of data passage should be detectable.

Ex-filtrate In general malware is known to extract information from the infected host and ex-filtrate it to controlling infrastructure [Stone-Gross et al., 2009]. The same should be expected from malicious software agents in a malicious multi-agent system. The information ex-filtrated from hosts could include: banking information, credit card numbers, passwords, proprietary secrets, or anything else storable on a machine.

A malicious software agent capable of ex-filtrating data is typically referred to as a *trojan*, since trojans allow the attacker to log into the victim machine through the trojans interface, and allows the attacker to send and receive information [Rieck et al., 2010]. However, as malicious software agents become more autonomous, less and less human interaction from an attacker will be required - until such a time that the malicious multi-agent system expresses an interest in a particular type of information and the malicious software agents are automatically tasked to retrieve the data.

As in propagation, the malicious software agent must send data to achieve exfiltration, as well as receive taskings from the malicious multi-agent system. The interactions between the malicious software agent and the malicious multi-agent system should be detectable by Multi-Agent Malicious Behaviour Detection.

Update Various existing malicious multi-agent systems are maintained by attackers by means of downloaded updates or patches sent from controlling infrastructure. Often updates may be identified by an unusual download of an executable file [Leder and Tilmann, 2009]. While malicious software agents may be able to self-replicate, in order to continue to evolve external updates must be sent from the malicious multi-agent system to the individual malicious software agents. Updates enable the malicious software agents to improve their ability to avoid detection, or add novel capabilities. Conficker is a rich example of a malicious multi-agent system comprised of various versions of malicious software agents that are routinely upgraded (Section 1.2.2).

Updates are another behaviour that involves communications between malicious software agents in a malicious multi-agent system. As with propagation, updating should be identifiable by the transfer of data from an external malicious multi-agent system infrastructure out to various groups of malicious software agents. However, updates are likely to happen less regularly than propagation attempts and may be difficult to predict.

Mimic In order to avoid detection, malicious software agents require the capability to mimic existing traffic on a network. More sophisticated malicious multi-agent systems will be capable of achieving a higher degree of detection avoidance. Simple mimicking techniques include using DNS for beaconing, or delivering exfiltrated document in HTTP post messages. If the individual malicious software agents are sophisticated enough to blend into their network environment by mimicking the common protocols their detection will be difficult. From the perspective of Multi-Agent Malicious Behaviour Detection, the mimic behaviour requires attention to benign protocols and mechanism for identifying abuse of those protocols.

1.6 Hypothesis and Research Questions

Now that I have described the challenges associated with detecting malicious multi-agent systems and introduced motivations for the increasing complexity of malicious multi-agent systems, I will provide the principal hypothesis and research questions that this thesis will address.

1.6.1 Hypothesis

There is a trend demonstrated by the discussion so far that malicious software agents are evolving increasingly sophisticated techniques to evade network defenders. While individual malicious software agents might be of limited intelligence, the interaction of a collection of malicious software agents as a malicious multi-agent system (perhaps supported by human operators) presents a difficult problem for computer security. The problem is comparable to work in competetive multi-agent systems in the field of artificial intelligence, and treating malicious software agents as artificially intelligent autonomous agents, like those commonly studied in the field of artificial intelligence, is worthwhile. Further, treating groups of interacting malicious software agents as a malicious multi-agent system is justified.

1.6.2 Research Question 1

How can advances in artificial intelligence, such as machine learning and multi-agent systems, be applied to computer security to increase the success of detecting and mitigating malicious multi-agent systems?

1.6.3 Research Question 2

Given that malicious multi-agent systems are expected to evolve, including changing their communications to adapt to target networks, how can advances in artificial intelligence, such as machine learning and multi-agent systems, be used to improve discovery of novel and/or evolving malicious multi-agent systems?

1.6.4 Research Question 3

Given that the complexity of computer network security requires some level of human expertise, how can a methodology involving a collection of semi-autonomous Multi-Agent Malicious Behaviour Detection Agents improve the capability of network defenders to detect and monitor sophisticated malicious multi-agent systems? How can such a system limit the cognitive load on the network defenders while still providing increasing value in malicious multi-agent system detection capability?

1.6.5 Research Question 4

What methods exist to ensure that as research in multi-agent systems and machine learning progress, those benefits are realized in computer network security and employed against malicious multi-agent system detection?

1.6.6 Discussion

The hypothesis follows from earlier sections in this Chapter and the rest of this research will reinforce the validity of the hypothesis. The motivations and justifications for each research question follow from the discussion to this point, and will be addressed throughout the rest of this thesis. To elaborate on reasearch question 4, given the evolving nature of this problem any proposed solution that cannot be readily adapted to take advantage of new techniques for detection will quickly become obsolete. This research emphasizes the importance of designing and implementing a Multi-Agent Malicious Behaviour Detection framework that is extensible such that any improvements in machine learning and multi-agent systems can be readily adopted in the Multi-Agent Malicious Behaviour Detection framework. Additionally, providing mechanisms to abstract details of specific processes away in a modular format so that further research into computer security can be introduced into the system without significant effort is important to ensure the research here continues to be relevant for some time after this work is published.

1.7 Summary

The research presented here deals with the detection of malicious multi-agent systems and presents a novel framework that furthers research in multi-agent systems, computer network security and human machine interaction while taking advantage of advances in machine learning techniques. Multi-Agent Malicious Behaviour Detection blends real-time detection with human level analytics from network defenders to further improve detection capability. Multi-Agent Malicious Behaviour Detection takes advantage of existing behaviours defined by network defenders using various analytics, like observing malicious multi-agent systems or direct interaction with malicious software agents. Multi-Agent Malicious Behaviour Detection consists of a custom multi-agent system leveraging machine learning techniques to recognize sets of additional behaviours shared by a variety of malicious multi-agent system agents. The shared behaviours are further used to detect previously unknown instances of malicious multi-agent systems as well as individual malicious software agents. The detectable behaviours I will consider are: beacon, deny, propagate, ex-filtrate, update, mimic.

The next chapter will discuss further background in network security, describing the range of computer network attacks and introducing terminology used throughout the rest of this thesis. Chapter 2 is important for those who are new to computer network security, or are seeking a refresh on the topic. Chapter 3 describes a variety of technologies and methods currently deployed for detecting malicious behaviours and reviews other similar work in the field of both artificial intelligence and computer security. Chapter 3 includes discussion on machine learning and multi-agent systems as well. Chapter 4 presents the Multi-Agent Malicious Behaviour Detection Architecture. Chapter 5 describes the design and implementation of the Multi-Agent Malicious Behaviour Detection Architecture. Chapter 6 describes the experiments performed to evaluate the architecture implemented in Chapter 5 and discusses their results. Finally, Chapter 7 provides an analysis of the results and the conclusions that can be drawn from this research, along with some indication of useful future work in this area.

Chapter 2

Background

2.1 Overview

Computer Security is a rich field that can be overwhelming if the proper background is not provided. This chapter seeks to provide additional background to help improve the overall understanding specifically of computer network security. While those already well versed in computer network security may choose to skip this chapter, I encourage readers who are new to the area to take the time to go over the next few pages where I introduce some terminology, types of detection systems, traditional technologies commonly deployed and finally end with a description of two detection methods. Chapter 3 expands on computer security research and will introduce artificial intelligence, machine learning and multi-agent systems.

2.2 Terminology

This section presents some common terminology used in computer security. Some terminology varies slightly in usage throughout the many sub-domains in computer security. Here I present each item within the context of this thesis.

2.2.1 Intrusion Types

There are many different types of system intrusions. Most, however, fall under one of six broad categories: worms [Moore et al., 2003], viruses [Chen and Robert, 2005], trojans [Livadas et al., 2006], scans [Whyte et al., 2006], botnets [Gu et al., 2008b] and zero-day exploits Crandall et al. [2005]. In the context of this research a malicious multi-agent system could involve any of these techniques. Here I give a brief explanation of each of these.

Worm The primary characteristic of a worm is its ability to self propagate, typically from one machine to another [Moore et al., 2003]. The worm executable will transfer itself from one victim to another across the network by exploiting some vulnerability in the target machine. A typical side effect of worm activity is a disruption of service due to network congestion as the worm attempts to spread to as many machines as it can reach. In the worst case, all machines are vulnerable and the worm will propagate exponentially. Legitimate traffic is either severely delayed or dropped as the network becomes saturated with malicious worm traffic. A worm may also contain a malicious payload, acting as a transportation mechanism for a virus or a trojan.

An example of a fairly straightforward worm is Slammer [Moore et al., 2003]. According to Moore et al. [2003], Slammer was able to spread to 90% of vulnerable machines in roughly 10 minutes, illustrating a simple but effective propagation method. The worm exploits a buffer overflow vulnerability in Microsoft SQL products. Because the vulnerability could be easily tested remotely with a single payload smaller than 400 bytes, Slammer's propagation quickly saturated network links. The worm pseudo-randomly chooses IP addresses and sends the exploitation payload to as many IP addresses it can generate before being detected. Aside from delivering the payload to exploit the target machine and propagating, there does not appear to be any other functionality in Slammer.

Virus A virus embeds, or infects a file on the host machine [Chen and Robert, 2005]. Its primary method for propagation involves infecting other files. As users distribute infected files to other machines, through peer to peer networking, portable digital media distribution, email attachments and embedded links, or traditional file sharing the virus spreads. Since the Virus embeds itself into otherwise non-malicious files, its spread is dependent on files being copied to other computers or by infecting files on network shares [Chen and Robert, 2005].

Trojan A trojan, sometimes referred to as a backdoor, is an application that allows a user to have access to a remote machine, typically without the victim knowing and often by covert means [Livadas et al., 2006]. Once installed, many trojans listen for connections to the victim machine on predefined ports and when a connection arrives, the attacker can use the connection to perform a number of tasks on the victim machine. Often trojans are referred to as remote administration tools, boasting legitimate functionality. However, their functionality is routinely abused and many popular trojans are considered malware. A trojan is typically installed by another program, often referred to as a *dropper*. A trojan may be the malicious payload in a worm, attached to an email in a content delivery attack (see Section 2.2.2), or downloaded when a user follows a malicious URL.

Subseven [Livadas et al., 2006] is a well known trojan that has been active since 1999. Although marketed as a remote administration tool, Subseven has been used widely as a trojan for mischief and criminal purposes. Subseven typically opens port 27374 and communicates using an IRC channel or ICQ, allowing a controller to connect and manipulate the infected machine. It provides the controller with various functionality such as the ability to control the mouse, webcam and microphone. It allows the controller to take screen shots, sniff traffic, read/write files, and log keystrokes. Finally, Subseven can instruct the machine to participate in a DDoS attack [Poulsen, 2001], a deny behaviour as discussed in Section 1.5.4.

Scan Although not an attack per se, a network scan often occurs before a specific attack [Whyte et al., 2006]. The attacking host will send the victim host some number of connection requests to a variety of ports on the victim machine. The attacker uses the responses to the connection requests to determine if a particular service is running on the victim. Once a list of ports with services is identified, further scans are used to determine if the discovered services are vulnerable to attack. For example, an attacker might start by sending a *syn* packet to port 80 and get a *syn ack* response from the victim. The attacker may then send another syn packet and try to establish a full connection with the server by completing the TCP three way handshake. Once the connection is established the victim will likely return a banner indicating the version of the server as well as other key information that the attacker can then use to determine if the victim is vulnerable to attack. Scans are often performed in bulk, where an attacker might scan an entire network in search of vulnerable servers. Much research has been performed in order to identify such scans, assuming that a scan is an indicator of a imminent attack [Whyte et al., 2006].

Botnet A botnet is an example of a malicious multi-agent system (see Section 3.4) and is the topic of a significant body of computer security research [Gu et al., 2008b]. Botnet victims are organized into a network of similarly compromised machines ready to accept commands from a botmaster. Traditionally botnets are organized in a hierarchy, with bot machines at the bottom, capable of accepting commands from a layer of controllers that are ultimately controlled by a botmaster. The botmaster directs the actions of the bots by sending commands to them through controller hosts. Often the hosts communicate through an IRC channel, or by way of HTTP [Gu et al., 2008b]. There are also P2P botnets, such as the Storm [Grizzard et al., 2007; Holz et al., 2008] and Nugache [Dittrich and Dietrich, 2008] botnets, which communicate with peers to get lists of commands [Kang et al., 2009]. Machines become part of a botnet post infection by a number of available means, such as social engineering [Lauinger et al., 2010], content delivery [Chen, 2003], application vulnerabilities [Andersson et al., 2004;

Jimenez et al., 2009; Clarke, 2009], operating system vulnerabilities [Jimenez et al., 2009; Clarke, 2009] and password guessing [Holdings, 2010; Zatko, 2009]. Currently, identifying botnets often focuses on detecting hosts involved in denial [Lawniczak et al., 2009], ex-filtration [Rieck et al., 2010] or propagation using a variety of techniques. Each of these corresponds to the behaviours described in Section 1.5.4.

Zero-Day Exploit A zero-day exploit is a vulnerability in an application that can be exploited by malware developers to attack a host machine that is unknown to the application developer. Once the application developer is aware of the vulnerability, the zero-day exploit is no longer a zero-day, as the application developer now has a chance to fix the vulnerability in the target application. Zero-day exploits are dangerous as they are unlikely to be identified by most anti-virus or intrusion detection systems [Crandall et al., 2005].

2.2.2 Attack Vectors

There are a number of different methods used to exploit information technology. The following subsections describe some of the most common.

Social Engineering Social engineering aims at gaining access to networks through taking advantage of people who already have access. Social engineering is described in a number of research papers as a an attack vector, such as Chen and Robert [2005], Dittrich and Dietrich [2008], Holz et al. [2008], Rieck et al. [2010], Wurzinger et al. [2009], Stamos [2010] and Deibert and Rohozinskli

[2009]. For example, an attacker can call a victim pretending to be a representative of the victims network technical support team. The attacker would then ask the victim a variety of questions and try to get the victim to reveal passwords, or network information that could facilitate the attackers approach on the network. Social engineering also generally includes such techniques as dumpster diving (where the attacker tries to find information about the victim network by searching through the trash bins near the physical network location) and email fraud. Email fraud, when not used in the context of content delivery, also aims at trying to convince the victim to reply with a password or reveal additional information. Computer security techniques for preventing social engineering attacks rely heavily on informing the end users about the risk of sharing information with outsiders.

Included in social engineering attempts are automated methods for eliciting users to following links. The links can be sent via spam emails, randomly posted in chat channels, or presented to users in phishing attacks. Phishing attacks are especially effective, as the malicious user will often present the victim with a convincing looking web-page or a pop up similar to a web-page they frequent. The fake website will try to convince the user to either follow a link or give out personal information. Lauinger et al. [2010] also describe a man in the middle attack, where a bot forwards messages from one person to another in a chat room, and once the users appear to have a genuine conversation the bot looks for opportunities to insert malicious links that one of the participants in the conversation are likely to click on.

Content Delivery Content delivery typically involves an attacker sending emails out to targets with an attachment containing either a virus, a trojan, a worm or some combination of those. The attacker may be expressly targeting a particular user by trying to send a trojan exploitable at a later time, or a worm might attach a copy of itself to an email and send the copy to people residing in the victims email contacts, like in the case of the ILOVEYOU worm [Chen, 2003].

As users become more educated about the dangers of opening attachments, social engineering becomes more key to the success of a content delivery attack. To that end, attackers will often try to fake legitimate looking email addresses with convincing content, or send emails from accounts that exist in the target's contacts list.

Application Vulnerability A number of applications are vulnerable to exploitation due to various reasons. Often a remote machine's interactions with a vulnerable machine can allow the attacker to obtain privileged access to the victim machine. For example, in 2000 a vulnerability in wuftp allowed users logging into the ftp server anonymously to supply a malformed password. This in turn was used to cause an overflow in the password buffer and allowed the attacker to exploit the buffer by inserting malicious code for spawning a root shell [Andersson et al., 2004]. Vulnerabilities such as these crop up on a regular basis, some of the most popular ones being SQL injection attacks against web servers and various exploits targeting Adobe PDF and Microsoft document formats [Jimenez et al., 2009; Clarke, 2009]. Many of the vulnerabilities appear in features intended to improve the service or product, but in fact also increase the victims' attack surface by introducing insecure code.

Operating System Vulnerability Operating system vulnerabilities used to be very common. The core programming of the operating system itself would contain exploitable errors in code. Many operating systems have implemented enhanced security in an attempt to reduce the number of vulnerabilities; for example device driver signing, non writable memory locations, chroot jails, etc. Given that operating systems, by their nature, involve a very large amount of code and functionality, with the additional requirement of inter-operating with third party applications, ensuring an operating system cannot be exploited is a complex problem [Jimenez et al., 2009; Clarke, 2009].

Password Guessing Password guessing is a brute force approach that attempts to raise the attacker privileges by simply guessing an account password over and over again. Traditional password guessing relies on dictionary attacks, where the password guesser will try every word in a predefined dictionary, often starting with common passwords and then moving on to uncommon passwords. Password guessers have taken advantage of a number of user weaknesses in passwords, such as common words, adding numbers to the end of common words, years of birth, etc. Once a set of common passwords, and passwords derived from common passwords is exhausted, many password guessers will begin a complete search of all possible combinations of either alphabetic, alphanumeric or alphanumeric with additional symbols. Exhausting the space of all possible passwords is a purely brute force approach, but given enough time this approach will always succeed. Many operating systems and other applications requiring password access allow administrators to set a fixed number of password attempts before they lock a user account out of the system. This feature has limited the usefulness of password guessing attacks in real-time. However, there still exist applications that allow offline password guessing that work against password files, e.g. lophtcrack [Holdings, 2010; Zatko, 2009].

2.3 Detecting Malicious Software

In general there are two broad approaches for deciding where detection of malicious software should take place, depending on the physical and logical location of the detecting software agent, referred to here as a *detection agent.* It can be done in a distributed manner, with detection agents on each host responsible for monitoring the target hosts' behaviour, or the detection agent can be positioned somewhere in the network where it is responsible for intercepting and monitoring traffic destined for several different hosts. These two approaches are termed *host-based* or *network-based* intrusion detection [Kabiri and Ghorbani, 2005]. Both approaches have advantages and disadvantages, and these are discussed below. Additionally, detection of malware can occur in-line, offline, or passively. The choice of which method to use can impact the security of the network by limiting either the variety of methods used for detection or the speed/type of response to a detection event.

2.3.1 Network-Based Detection

The role of the network detection agent is to observe the traffic at one or more layers of the TCP/IP model (application, transport, Internet or link). A network detection agent could operate at any layer of a network-based off the OSI model. However for simplicity my work will only consider the TCP/IP model. The network detection agent collects some subset of the data it is able to observe, and in the simplest case, classifies it as either good or bad. The network detection agent may be responsible for one or more follow-up actions, such as logging a representation of all traffic, logging bad traffic in the form of alerts, collecting traffic for post analysis, or blocking bad traffic from entering the network [Dreger et al., 2008; Chen et al., 2010]. Network-based detection requires each detection agent to be positioned at choke points throughout the network. Choke points typically reside at routers/switches where traffic aggregates and is then distributed out to the rest of the network. On broadcast networks, where every communication is broadcast and each host can identify which messages belong to them, the network detection agent can reside anywhere in the network and still receive all traffic. When positioned correctly across the network, all traffic in to and out of the protected network must pass through a network detection agent before arriving at its intended destination.

One advantage of network-based detection is that it provides the network administrator with a broad view of what is occurring on a network without having to query every individual machine. Installing and managing sensors on individual machines may be prohibitive due to resource constraints. Additionally, actions can be taken on network traffic before it reaches the individual host. A host may be vulnerable to a particular attack, but a network detection agent that is not vulnerable to such attacks can either stop the traffic from getting to the vulnerable host or notify the system administrator that such traffic is destined to the host.

Network-based detection agents must determine how the destination host will treat individual packets, and therefore a major design decision is whether the sensor will operate on individual packets out of context or act on sessions of reassembled packets. Network detection agents are more complex when designed to process the application or reassemble the transport layers of the target traffic. However, malware that operates at the application layer may be difficult to detect at lower levels of the TCP/IP model [Handley et al., 2001]. For example, at the IP layer a network-based detection agent can use a blacklist to limit traffic coming into the network from "bad" IP addresses. To operate at the TCP or UDP layer the network detection agent must keep track of ports that are associated with each IP address. The feature set has increased from 2 unique features to 5, commonly referred to as the 5 tuple. The 5 tuple consists of source IP, source port, destination IP, destination port, and protocol, where the protocol is typically UDP or TCP. At the transport layer the network-based detection agent can filter protocols based on port and can also begin filtering on the contents of individual packets. Operating at the application layer requires maintenance of significantly more state information. Each TCP or UDP packet can be recombined with the other packets in the same stream to build the session, containing the complete message being sent to the destination host. The network-based detection agent must be able to reassemble the packets into the same session that the destined host would reassemble the packet in order to get the message accurately. Handley et al. [2001] have shown that network-based detection agents can be exploited by forcing the application layer to be fragmented across the transport layer in such a way that reassembly of the packets will result in different sessions depending on the operating system performing the reassembly. The network detection agent is tricked into reassembling the wrong stream and misses a feature of the stream that would have resulted in classifying the stream as "bad".

Finally, network-based detection can suffer from lack of information on how application layers should be decoded. If the application layer is encrypted or otherwise encoded using a shared secret between the source and destination of the communication, then without the shared secret or decoding information, the network-based detection agent will be unable to view the information being passed between the two hosts and will therefore be unable to make an accurate classification of the traffic. Certain network policies may prohibit encrypted communication or require encrypted communications be registered with the administrators. However, as encryption becomes ubiquitous in network communication, network-based detection will become more difficult [Goh et al., 2010, 2009].

Specific examples of network-based detection agents are Snort [Papadogiannakis et al., 2010; Snort, 2010], Sax2 [Trabelsi and El-Hajj, 2010; Sax2, 2010], Bro [Paxson, 1999; Vallentin et al., 2007; Flaglien, 2007] and Shoki [shoki, 2010]. There are a variety of other network-based detection agents.

2.3.2 Host-Based Detection

Host-based detection requires a detection agent deployed to each host. From the host the sensor is capable of observing network traffic, as well as processes running on the host. Ideally, each network communication is associated with a specific process or more generally an application. In addition, a sensor may have access to the memory space of each process as well as access to read and write to the file system on the host. Host-based detection is therefore capable of performing a number of tasks that network-based sensors are not. For example, if a host is the victim of a trojan, the network sensor must rely on the network traffic to determine the existence of the trojan. If the network traffic is encrypted or otherwise encoded, the trojan might go unnoticed. The host-based detector can monitor the list of running processes, and from there determine if a trojan is running. Upon discovery the host-based detector may be able to terminate the process and notify the administrator of the existence of the trojan, so that it can be cleaned from the system [Molina and Cukier, 2009; Jiang and Wang, 2007]. The host-based detector may have the capability of removing the trojan itself, if it is able to determine what file the trojan resides in. Maintained host-based detectors are effective at taking care of a single host. Many anti-virus vendors currently offer services like the ones explained above, as typically anti-virus is a form of host intrusion detection [Kolbitsch et al., 2009].

Host-based sensor management must scale to the size of the protected network. Each host should have its own instantiation of the host-based detector and each of those detectors must be maintained up to date. Many private sector vendors offer managed services for host-based sensor management, for example [Mcafee, 2010; Symantec, 2010; Kaspersky, 2010].

The host-based sensor is susceptible to attack, which may render the sensor ineffective. If the administrator relies on the integrity of the host-based sensor, and that sensor is compromised then the machine itself becomes compromised. There are several worms and viruses that attempt to terminate anti-virus processes. Once the anti-virus is disabled the machine is vulnerable [Molina and Cukier, 2009]. Additionally, virus/worm writers can acquire access to the same commercial host-based sensors that are available to their targets. By testing the detection capability of these commercially available and managed host-based sensors, virus/worm writers can modify their malware until it is not detected.

There are several host-based intrusion detection systems available, examples include Swatch [Hansen and Atkins, 1993; Swatch, 2010] , Snare [Zhu and Hu, 2003], Sentry Tools [Tools, 2010] and Log Surfer [Surfer, 2010]. Host-based intrusion detection has existed for decades [Lane and Brodley, 1998; Lee and Stolfo, 1998]. However, recent research on host-based intrusion detection include [Hu et al., 2009; Hugelshofer et al., 2009].

Adopting a combination of host and network-based detectors allow administrators to gain the benefits of both types of detectors.

2.3.3 In-line Detection

The main feature of in-line detection is that all network traffic must pass through the detector before it continues on to the rest of the network. In this case, each packet or reassembled session must be analyzed and classified in real-time as the detector processes them. The packets may be forwarded one at a time, or the sensor may attempt to reassemble sessions on behalf of the host, perform detection on the session and then forward the session to its intended destination. In any case, the detector will impact the flow of the traffic. Ideally the detector's impact will not severely limit the usability of the network. There are some very simple detection techniques performed routinely by network detector systems in real-time, such as port filtering and IP address blacklisting. Other more complex tasks, such as matching strings, calculating entropy and decoding ciphers, can seriously impede the network flow [Scarfone and Mell, 2007].

If in-line detection is achieved and the impact on the network is manageable, the detection agent gains the capability to take action on traffic before it gets to the host. For example, the detector can actively block specific connections from taking place, reset malicious sessions or redirect traffic to follow on processes. In-line detection provides the sensor with flexibility with respect to taking action against malicious traffic, but limits the detection capability of the sensor by requiring it to detect and respond in real-time.

In-line detection may allow the detector to become a target of attack. Given that the in-line detector is potentially accessible to would-be attackers, any vulnerability in the detector itself may be exploited by an attacker. Once an attacker controls the in-line network detector they may perform a variety of malicious tasks, such as modifying black lists, denying service, or modifying traffic. Another attack on in-line detection involves an attacker having knowledge of a subset of the in-line detector's rule set. An attacker could potentially force a denial of service by sending traffic they know will trigger more alarms then the in-line detector can cope with, therefore degrading network performance.

2.3.4 Passive Detection

Passive detection is a slight variation on in-line detection. The detection agent gets its own copy of all traffic entering and leaving the network from a particular point in the network, sometimes enabled through a spanning port on a router or a network tap. The detector is then not accessible by any outside machine, since it is not connected directly to the network. The advantages of this configuration are: smaller attack surface for would be attackers, process traffic may not be in real-time, but can lag as long as during slow traffic periods it has the capability to catch up to current traffic, it does not directly effect the usability of the network it is protecting. The disadvantage is its limited ability to take action against traffic in real-time [Scarfone and Mell, 2007]. There is technically no delay between the time when the detector gets a copy of the traffic to when the host where the traffic was destined gets its copy of the traffic, therefore the detector can not prevent the connection from being established. However, if the passive detection is capable of performing detection in near real-time, it may be able to reset connections through some other means before the attacker completes its session.

2.3.5 Offline Detection

Offline detection is akin to network forensics. However instead of being in response to an intrusion, offline detection involves analyzing stored traffic with the intention of finding attacks. Offline detection typically requires a device on the network to write all data to hard disk where analysts can apply algorithms to the traffic. Many intrusion detection algorithms that require intense processing or large memory requirements have to post-process traffic since it cannot be done in real-time.

The main disadvantage to offline detection is that there is a delay between the attack and response. Once an attack is identified too much time may have passed to effectively respond to the attack. For example, determining that a DDoS attack occurred the day after it finished is not efficient for preventing that particular attack. Offline detection does enable prevention of future attacks even if they are identified during post processing. Identifying that a botnet is expanding on the network still allows network administrators to find and remove the existing trojans. Especially if finding them cannot be done in real-time due to processing limitations.

2.4 Traditional Technologies

This section describes four traditional computer security technologies that are common in computer networks today: firewalls, intrusion detection systems, intrusion prevention systems and honeypots.

2.4.1 Firewalls

A firewall is an example of a simple in-line network security device. Traditionally a firewall is designed to work at the Internet layer by filtering traffic by IP address or the Transport layer by filter traffic according to the port of the traffic and its transport protocol. In effect a firewall operates against a five tuple and either allows or denies traffic in and out of the network depending on the five tuple. Firewalls are meant to be a fast response first line of defence against Internet misuse and are common in most networks [Chapman and Zwicky, 1995]. Firewalls used to be implemented as stand alone appliances or machines, but are now integrated into many home as well as enterprise class routers. Host-based firewalls are integrated with many popular end user operating systems, for example Windows, Mac OS and Ubuntu have integrated firewalls [Al-Rawi and Lansari, 2007; Peisert et al., 2010; Kurland, 2010]. Host-based firewalls, in addition to basing access to the machine by five tuple, are also capable of limiting access to the Internet by applications installed on the host machine. The relative simplicity of firewalls make them ideal for deploying at multiple levels of the network. However, their simplicity also makes them susceptible to exploitation. Firewall rules are updated based on blacklists, requiring malicious IP addresses to be identified before they can be blocked. While the space of IP addresses is finite, attackers can still originate from novel IP addresses with each new attack. Victims offering services to a large number of consumers cannot block large subsets of the IP space without impeding the victim's ability to serve the consumer base. For a firewall to be truly effective, the administrator must know ahead of time exactly which IP addresses are the ones that can be trusted to interact with the victim's machines.

Finally, since firewalls are typically support simple allow/deny settings based on five tuples, any learning or modification of the firewall must be done from some other application or device.

2.4.2 Intrusion Detection Systems

Intrusion detection systems (IDS) are designed to be more complex than firewalls, with the intention of observing traffic and then logging alerts when traffic matching a rule or heuristic is triggered. An IDS traditionally inspects further into a packet than a firewall (see Section 2.4.1). The IDS engine examines the IP and TCP/UDP header features, such as source and destination IP addresses, ports, TCP flags, etc., acting at the second and third layers of the TCP/IP protocol stack. Additionally, an IDS may look at the data section of an individual packet, or sessionize several packets and match its rules/heuristics on the application layer [Scarfone and Mell, 2007]. The capabilities of an IDS are dependent on the placement of detection agents as described in Section 2.3.

2.4.3 Intrusion Prevention Systems

Intrusion Prevention Systems (IPS) are an extension to the IDS, and were originally designed to allow dynamic modification of firewalls to block ongoing or predicted attacks. Where an IDS is a passive device, monitoring network traffic and generating alerts, the IPS monitors traffic and actively blocks traffic, either by modifying an existing firewall or by itself acting as a firewall. Low false positive rates are desirable for a system that blocks traffic dynamically without user intervention, and posses a greater risk of blocking legitimate traffic if the rules/heuristics are unable to accurately describe the malicious activity without producing false positives [Scarfone and Mell, 2007].

2.4.4 Honeypots

In the context of computer network security, the *honeypot* is a monitored machine that is intentionally left vulnerable to attack by malicious software. Such vulnerabilities may include the absence of a firewall, easy-to-guess passwords, unpatched software, older operating systems, etc. The goal of the honeypot is to attract malicious attackers. The honeypot administrator can then study any malware that infects the honeypot to learn how to better protect other computer networks. Groups of honeypots are sometimes referred to as *honeynets*. Since there is a significant amount of automated victim discovery, a honeypot has the potential to be compromised by automated as well as supervised attacks. There are, however, several disadvantages to honeypots. First, a large number of attacks require user intervention. Many attacks require content delivery (e.g. a download of an infected webpage, or opening of an infected attachment). With no one to access a document infected with malware, the honeypot may never be infected. Second, once a honeypot is infected it may participate in malicious behaviour. There are ethical issues with respect to allowing a machine to participate in malicious behaviour that may cause damage to other systems. Third, honeypots require patience, and the infecting malware may not be easily identified, requiring a significant amount of work for it to be effective [Baecher et al., 2006; Provos, 2004; Balas and Viecco, 2005].

2.5 Detection Models

This section provides details on three detection models that will be incorporated into the architecture of the system described in Chapter 4: misuse detection, anomaly detection and heuristic detection.

2.5.1 Misuse Detection

Misuse Detection identifies malicious behaviour based on a set of predefined rules or signatures [Chen et al., 2007]. The rules generally identify specific instances of known attacks. For example, a misuse detection rule may identify traffic on a specific port containing a specific byte sequence as belonging to a particular bot. Consider an example rule from the Bleeding Snort Virus rules [Snort, 2010]:

> alert tcp $HOME_NET any -> $EXTERNAL_NET 25 (msg: “BLEEDING-EDGE VIRUS - Bugbear@MM virus in SMTP”; flow established; content: “uv+LRCQID7dIDFEECggDSLm9df8C/zSNKDBBAAoGA0AEUQ+”; reference:url,www.symantec.com/avcenter/venc/data/w32.bugbear@mm.html; classtype: misc-activity; sid: 2001764; rev:4;)

A more in-depth discussion of the Snort rule structure will be addressed in section 2.5.3. The signature above looks for TCP traffic leaving the home network going to an external network over port 25 and having the specific content string above. The rule identifies traffic from a known intrusion attempt. If the bugbear virus is ever changed and even a single byte of the content is modified, the signature will no longer detect the bugbear virus and will require modification in order to detect any new variation. This is a critical weakness in traditional signature detection, since a number of malware distribution mechanisms will obfuscate their code with the intention of evading signatures. Some of these attempts include applying XOR masks to the data [Stone-Gross et al., 2009; Baecher et al., 2006], dynamically injecting code [Wang et al., 2006], implementing custom decoding routines [Wang et al., 2006], or encrypting/packing payloads [Gu et al., 2008a].

Many traditional IDSs are based on misuse detection. Security analysts write rules as new attacks are published, potentially leaving a time frame between updates where systems are vulnerable. Misuse detection typically requires manual maintenance to keep the rule lists deployed to sensors up to date, essentially responding to known threats. This constant update requirement is one major disadvantage of misuse detection based systems [Zanero and Savaresi, 2004a].

2.5.2 Anomaly Detection

Anomaly detection is a fundamental aspect of computer network security, and in general identifies anomalies in a given data set. Anomalies in intrusion detection can be characterized by several deviations from expected activities, such as deviations in network traffic or process behaviour on a host. When dealing with network traffic, the goal is to classify a given sample of traffic as either an anomaly or normal, and more specifically as either malicious or benign. Anomaly detection is often described as a classification problem[Zanero and Savaresi, 2004b; Wang et al., 2009; Teng et al., 2010; Sun and Wang, 2010; Khor et al., 2009]. However, if plenty of examples from every class are not available when designing the system, the anomaly detection engine will only be able to model normal behaviour and then infer anomalies from those inputs that are determined different than the data already observed, as described in the papers referred to above. Classification when examples of each class are available and labelled is termed *supervised learning*, while *unsupervised learning* or *novelty detection* refers to data sets where labels are not provided with the data or there does not exist examples of all of the classes in the data set [Yeung and Ding, 2003]. Essentially, the system has to learn to assign each sample to a category as malicious or benign without a priori labels [Portnoy, 2000]. Unsupervised learning is desirable as it requires less involvement of the end user. There are a number of previous Intrusion Detection Systems that have attempted to use unsupervised learning [Zanero and Savaresi, 2004a]. For more information see Chapter 3.

2.5.3 Heuristic Detection

A *heuristic* defines a set of characteristics that are associated with an anomaly. However heuristic detection is more general than misuse detection [Dash et al., 2006]. For example, consider the type of traffic expected on port 80. Port 80 traffic typically consists of HTTP traffic, and one might expect that any other traffic on port 80 is an anomaly. Normal HTTP traffic should begin with an HTTP header, either a GET or a POST. A very simple heuristic is thus: *if a session on port 80 does not contain GET in the first content carrying packet of the session (i.e. after the TCP handshake), then mark it as an anomaly.*

A number of heuristics can be generated by observing network traffic and documenting exceptions for traffic. Typically heuristics are manually tailored by the system administrator of a network to suite the type of traffic they expect to observe. Heuristics can be represented as IDS rules in many cases, assuming that the grammar for describing the IDS rules is expressive enough to encompass the details of the heuristic. If the grammar is not expressive enough, it can lead to rules that do not encompass the entire meaning of the heuristic and generate false positives through low specificity.

Snort is an IDS with a grammar capable of identifying traffic by several traffic features, allowing rules to be written that map to user defined heuristics. The basic structure of the rule is:

> <alert type><protocol><ip address A><port A> <-> <ip address B><port B>(<option 1>: <arguments 1>; <option 2>: <arguments 2>; ... <option N>: <arguments N>)

The first 7 fields comprise the rule header. <alert type> allows the user to define types, for simplicity sake we'll consider it to be *alert*. The <protocol> describes the network protocol, such as TCP, UDP, IP, etc. It also allows two IP addresses and two ports with a directionality identifier between them. <- targets traffic from IP address B and port B to IP address A and port A, while -> targets traffic in the opposite directions, from A to B. Lastly <-> identifies traffic in both directions. The IP addresses can be either specific address, such as 192.168.15.1 or the term *any*, which signifies that any IP address can match the heuristic. Equally, the ports can be numbers from 1 to 65536, each representing the 2 byte port number of the traffic, or the term *any*, which matches all ports.

For example a rule header could be constructed as follows:

> *alert tcp 192.168.15.1 80 -> any any*

The rule header generates alerts on all TCP traffic originating from 192.168.15.1 on port 80 to any ip address and any port.

The next section of the rule contains a list of options and their arguments. The Snort manual describes three different option types: general rule options, payload detection rule options, non payload detection rule options, and post-detection rule options. It is outside of the scope of this discussion to address all of the options here. However, a sample of some of the common options, with a short description of each, follows. For more details please refer to the Snort user manual [Snort, 2010].

General rule options give information about the rule. Examples include the message (*msg*) that the rule displays when it fires, the signature id (*sid*) which uniquely identifies each rule and the priority if some rules require a faster response when they fire or special attention. Payload detection rule options enable deep packet inspection. One of the primary options for deep packet inspection is *content*. Content, as the name implies, performs matching against the content of the packet. The following rule will fire on all standard HTTP get requests:

> *alert tcp any 80 <- any any (msg: "Get request detected"; content: "GET "; sid: 12345;)*

Content can be modified with the payload detection rule options *offset* and *depth*, which specify how far to look into a given packet and where to start looking. Non-payload detection rule options specify the values for several protocol header fields, such as the packets time to live (*ttl*) indicating the number of hops the packet has passed through, the size of the packet (*dsize*) and the TCP flags (*flags*). Finally, post-detection rules offer optional actions to take once an alert fires. For example, the option *tag* will log a fixed amount of packets after a single packet triggers an alert. The option *session* extracts the session that the packet responsible for the alert resides in. The following example illustrates the potential complexity of the Snort rule grammar:

> *alert tcp 192.168.15.2 443 ->172.0.0.1 3443 (msg: "Example rule"; flags: PA; flow: established; content: "Some test string"; depth: 50; sid: 12345;)*

The rule above looks for TCP packets from 192.168.15.2 on port 443 to 172.0.0.1 on port 3443, where the push and ack flags are set, the flow is established and it contains the string "Some test string" in the first 50 bytes of the packet. When the signature hits on the packet it will send the message “Example rule” with the identification number 12345.

Heuristics are often realized in anomaly detection systems as rules, and generally speaking are generated by security analysts who study traffic to specify how to describe anomalous behaviours. Tasked with finding anomalies, the security analyst will typically use a number of tools and log files to search for anomalies.

2.6 Summary

This chapter provided a baseline of computer network security knowledge required to understand the remainder of this work. Now that the reader has security background in hand, I can begin examining the important prior research related to this work, including a more specific look at botnet detection, machine learning, multi-agent systems and a discussion on malware collection, feature extraction and finally, a brief look at the impact of encryption and virtual private networks.

Chapter 3

Literature Review

3.1 Overview

This research contains elements from the fields of autonomous agent development, computer security, and distributed artificial intelligence. Each of the fields named previously has been the topic of research for many years. The goal of this chapter is to introduce important research in areas related to this thesis and to differentiate this research from similar work. Included in this literature review will be a discussion of botnet detection techniques, the current state of machine learning in anomaly detection and an overview of multi-agent systems. I will also touch on research in plan recognition, malware collection and feature extraction. Finally, I will discuss how encryption impacts anomaly detection and the study of malicious multi-agent systems.

3.2 Botnet Detection

There are a several recent research endeavors focused on identifying botnets. They use a number of different techniques, and the following section details a subset of techniques that have been deployed for botnet detection.

Rieck et al. [2010] attempt to detect malware by focusing on "phoning home" techniques, and developed a system called Botzilla. There are three phases in Botzilla's malware detection. First it attempts to capture new malware using a variety of methods, including forensic analysis of security incidents, honeypots, honeyclients, etc. For more information on honeypots see Section 2.4.4. Once binaries for malware are collected, they are passed on to a repetitive execution phase. Each binary is executed several times in different environments within a sandnet. Executions involve modifying the host operating system, date and time, IP address, etc. Packet captures are collected for every execution. Botzilla performs signature generation using Bayesian techniques. Tokens are extracted from the network captures. Those tokens are assembled into signatures. The signatures are tested against live network flows and evaluated for their accuracy. The researchers claim that they achieved a 94.5% detection rate when deployed on a university network, and that their technique identifies novel malware automatically with no human intervention.

Gu et al. [2007] aim to monitor a network perimeter for coordination dialog and malware infections, using a system named Bothunter. The focus is on identifying the stages of a malware infection. Those stages are: inbound scanning, exploit usage, egg downloading, outbound bot coordination dialog and outbound attack propagation. Dialog event detection is based on Snort and enhanced with anomaly detection plugins SLADE and SCADE. The former performs a form of n-gram statistical payload analysis (discussed below), while the latter focuses on port scan analysis, weighting scan attempts based on the ports they target, their frequency and whether they succeed or fail. Given that many networks are attacked regularly, Bothunter attempts to associate outbound communication flows with botnet behaviour and then pair those outbound communications with intrusion attempts. Since Bothunter is intended to monitor a network's egress point (i.e. the point where traffic leaves a local network), they ignore DNS activity, local host modifications, and internal network propagation, assuming that those three characteristics are not reliable measures at the egress point. An important characteristic of Bothunter is its flexibility with respect to ordering of the coordination dialog. The authors ascertain that due to a number of conditions only some of the dialog may be captured, and work out a system of weights and thresholds that help to identify when a dialog is considered malicious. The Bothunter correlation engine tracks dialog events across time windows, aging off older events. When events occur that meet some threshold criteria, a bot profile is created. The criteria described in [Gu et al., 2007] are 1) an incoming infection warning followed by outbound local host coordination or exploit propagation warnings or 2) a minimum of at least two forms of outbound bot dialog warnings. Note that SLADE is tasked with identifying anomalous payloads, while SCADE is tasked with identifying inbound and outbound scans. However, Snort identifies known exploits, egg downloads and command and control traffic. Therefore, the signature engine requires human intervention to create and maintain signatures. Bothunter achieved a 95.1% true positive rate when tested in a honeynet environment. However, the results when deployed to a live University network and an institutional laboratory were not as conclusive as to Bothunter's effectiveness.

Botminer [Gu et al., 2008a] is an architecture designed to passively identify botnets by correlating command and control with activity traffic, independent of the former's structure or content. The authors ignore the initial infection, and instead focus on the ongoing traffic associated with the existing infection. The system is split into 5 components: C-plane monitor, A-plane monitor, C-plane clustering, A-plane clustering and Cross-plane correlation. The C-plane monitor collects network flow records. Each flow is identified by the following features: time, duration, source IP, source port, destination IP, destination port, number of bytes transferred in both directions, and number of packets transferred in both directions. The A-plane monitor attempts to identify activities associated with botnets, such as scanning, spamming, binary downloading, and exploiting. The A-plane is based on Snort, with the addition of the SCADE plug-in and an additional plug-in that tracks anomalous DNS queries and anomalous SMTP connections. The A-plane monitor also uses misuse detection for identifying binary downloads. C-clustering uses X-means clustering applied to a set of flows collected by the C-monitor over a specified period that have been converted to vectors and scaled. The A-clustering is based on activity type, followed by further clustering by features of the activities. Cross-plane correlation calculates a botnet score for each host that exhibits at least one suspicious activity. A threshold is applied to the score to indicate whether the host is likely part of a botnet or not.

Kang et al. [2009] describe a method for enumerating the hosts in a p2p botnet. They propose a Passive P2P Monitor (PPM), that passively participates in p2p botnets by implementing a bot with the capability to decode and understand the Storm protocol. However, they constrain the bot from actively participating in file sharing or sending malicious payloads. Instead, the bot is limited to routing messages in the botnet. They compare their approach to a crawler, which crawls the p2p network by first asking a starting node for a list of all the nodes it knows of, then asking the resulting nodes for lists of nodes they know of, etc. A crawl of the p2p network could potentially identify a large percentage of the network in a short amount of time with limited resource requirements. However, crawlers are limited in their ability to detect network address translated machines, or machines behind firewalls. The PPM uses a firewall checker (FWC) to attempt to identify which Storm IP addresses are originating from behind a firewall, and therefore improve their estimate of the botnet size. Kang et al. [2009] show that the PPM is capable of identifying more nodes than a crawler. However, the crawler may contain lists of nodes whose lifespan was too short for them to interact with the PPM. They conclude that any bot that sends approximately 200 messages will be identified by the PPM with a high likelihood.

3.3 Machine Learning in Anomaly Detection

In addition to techniques directed at botnets in particular, several machine learning techniques have been applied to anomaly detection, such as probabilistic detection [Ying-xu and Zeng-hui, 2009; Newsome et al., 2005; Khor et al., 2009; Farid et al., 2010; Farid and Rahman, 2010; Alpean et al., 2010]; artificial neural networks such as self organizing maps Polla et al. [2009]; Rhodes et al. [2000]; Portnoy et al. [2001]; Zanero and Savaresi [2004a]; Ramadas et al. [2003]; Cortada Bonjoch et al. [2002]; Lichodzijewski et al. [2002]; Kohonen [2001]; Labib and Vemuri [2002]; Girardin [1999]; Patole et al. [2010]; Yang et al. [2010], and multi-layer perceptrons [Vatisekhovich, 2009; Golovko et al., 2010; Ali et al., 2010; Mohamed and Mohamed, 2010; Sheikhan and Jadidi, 2009; Pinzon et al., 2010]; support vector machines [Mukkamala and Sung, 2003; Sung and Mukkamala, 2003; Gornitz et al., 2009; Pinzon et al., 2010; Teng et al., 2010; Sun and Wang, 2010; Wang et al., 2009; Yuan et al., 2010]; K-means [Faraoun and Boukelif, 2007; Chairunnisa et al., 2009; Zanero and Savaresi, 2004b]; and artificial immune systems [Greensmith et al., 2010; Twycross, 2007; Twycross and Aickelin, 2006]. The following sections will elaborate on the techniques techniques mentioned.

3.3.1 Probabilistic Detection

Probabilistic detection can be applied to anomaly/intrusion detection in two ways. First, a probabilistic algorithm processes the features of a traffic sample and uses a probabilistic classifier to compute the likelihood that the traffic is malicious (or more precisely, the likelihood that some series of events will occur). The lower the likelihood of a series of events, the more likely it is an anomaly. Additionally, the probabilistic algorithms can be applied to determine which features are important for detection by ranking the features by the probability that they will provide information related to anomalies. If a feature exhibits high information gain with respect to the occurrence of an anomaly, then it is ranked higher as an important feature.

Probabilistic intrusion detection typically has some foundation in Bayes rule, defined by the equation $P(h|D) = (P(D|h)P(h))/P(D)$, where D is the observed data and h is a hypothesis. From the equation,$P(D)$ is the probability of the occurrence of data D, $P(h)$ is the prior probability associated with hypothesis, $P(h|D)$ is the posterior probability and finally $P(D|h)$ is the conditional probability [Farid and Rahman, 2010]. When applying Bayes rules, the goal is typically to find the hypothesis with the highest probability given the observed data, or more precisely the maximum posterior hypothesis. Naïve Bayes classifiers are derived from the Bayes rule with the assumption that observed data attributes are independent of one another, and even though the assumption is often not valid in practice they typically perform well.

Ying-xu and Zeng-hui [2009] present a Half Increment naïve Bayes classifier, based on the traditional naïve Bayes classifier, to determine the probability that a given piece of compiled binary code is malicious. The authors enhance the performance of traditional naïve Bayes classifiers by introducing an evolutionary search into the feature selection process. They show that Half Increment naïve Bayes has a lower implementation cost than traditional naïve Bayes and Multi-naïve Bayes, as well as a faster execution speed and higher detection accuracy. Given that binaries are often transferred across a network, the same technique is applicable to network anomaly detection, assuming the transfer of certain binaries is in fact malicious.

Newsome et al. [2005] propose a type of probability based signature called a Bayes signature within the context of their Polygraph monitor. They score tokens in a flow and calculate the probability that the flow is a worm based on those scores. They use a threshold to determine if the resulting probability is high enough to classify the flow as a worm. In order to apply a naïve Bayes classifier, they assume that tokens identified in each malicious flow are independent of the existence of other tokens in the same flow. Applying probabilistic methods is considered to be more flexible than the exact match exact match typically required in standard misuse detection signatures, since the existence of a combination of tokens may indicate enhanced probability of an attack rather then certainty. Newsome et al. [2005] showed that generating signatures to detect polygraphic worms using probabilistic methods could be tuned to detect worms as well as some other signature detection algorithms. In their experiments, however, regular expressions could achieve similar, and in some cases better performance.

Khor et al. [2009] implement Bayesian algorithms to perform network intrusion detection. They choose a subset of selected features from the wider range of available features using empirical evaluation, and then build Bayesian networks based on the selected features. The feature selection process used in this research is further discussed in Section 3.7. They test their algorithm using the KDD99 intrusion set data. Once the features are selected, they compare the performance of a naïve Bayesian Classifier, a Learned naïve Bayesian Classifier and an Expert-elicited Bayesian Network. The Learned naïve Bayesian algorithm used existing search algorithms in addition to the conditional independence of the variables. The Expert-elicited Bayesian Network incorporates the views of domain experts in the construction of the network. They show that the classification accuracy varied depending on the classification of the attack, and of the three Bayesian approaches selected, none dominated. Bayesian networks obtained the highest classification accuracy with the Normal and DoS attacks, while naïve Bayesian Classifiers performed best at classifying probes and Remote to Local attacks, and the Expert-elicited Bayesian Networks achieved the highest classification on the User to Root category. They conclude that even though the performance of all three classifiers were comparable, the naïve Bayesian Classifier was the most efficient in terms of classification time on large data sets, while the Expert-elicited showed the most improvement when the there are a small number of attack samples compared to the data set size. For the exact numbers and more in-depth explanation for possible differences, see [Khor et al., 2009].

Farid et al. [2010] use naïve Bayesian trees in an attempt to reduce false positives in intrusion detection. They identify three issues in intrusion detection. First, unbalanced detection rates for different types of intrusion where the IDS is able to detect one type of attack with high accuracy like a denial of service but is less effective against network probing. Second, excessive false positives such as continually identifying benign DNS requests as malware beaconing. Third, redundant input attributes that do not provide additional value, but do impact the responsiveness of the learning algorithm. While their research focuses on the issue of false positives, they show that they can improve detection rates while also identifying important attributes. They also test their research on the KDD99 intrusion set data. Specifically, they rank the available network features using the naïve Bayesian tree, splitting nodes where there is maximum information gain. The actual rank of an attribute is determined by the minimum depth at which the attribute is tested. Important attributes are tested closer to the root, indicating that they were important in an early decision to split the data set into distinct sub classes. The authors modify the stock decision tree by adding standard naïve Bayes classifiers at the leaf nodes of the tree. They perform learning on training sets by iterative tree building based on the training data and updating the weights of the attributes depending on the depths of those attributes in the tree. See [Farid et al., 2010] for the complete pseudocode of their algorithm.

Farid and Rahman [2010] introduce the improved self-adaptive Bayesian algorithm (ISABA), based on the adaptive Bayesian algorithm. Given some training data set, the adaptive Bayesian algorithm derives a function to estimate the class conditional probabilities for each attribute value. Training is performed until classification of each example of the training set is successful or a target accuracy is achieved. An adaption process follows, where the classifier is tested against a set of test examples. The algorithm adapts by using the similarity between mis-classified test examples and training examples to modify the attribute weights with the formula $W_i = W_i + (S * 0.01)$, where S is the similarity. Adaptation is continued until the classifier correctly classifies all data in the test set. The ISABA adds the additional step of building a decision tree by information gain using the final updated weights of the training examples after all the test examples have been correctly classified. It also appears to slightly modify the adaptive update with $W_i = W_i + (S + 0.01)$. However this may be a typo, as there is a change from multiplication to addition and it isn't clear why it should be different from the previous formula. The authors' evaluation shows improvement in detection accuracy and some reduction in testing time. However the training time when compared to adaptive Bayesian algorithm is slower. When compared to naïve Bayes Classification the detection accuracy improves from 64% to 99.17% with respect to the User to Root attacks. Most other improvements are only marginal.

Alpean et al. [2010] introduce a probabilistic diffusion scheme consisting of a bipartite graph model that models probabilistic dependencies between a set of prototypes and a set of features that characterize those prototypes. Their goal is to detect malware and identify anomalies in smart phone usage by applying their Bayesian probabilistic model to the smart phone logs. Their algorithm contains elements similar to the Google adjusted page ranking algorithm. The algorithm is intended to be lightweight enough to run on a smart phone without taking up all of its resources. However, the research also demonstrates anomaly detection in a resource starved environment, which is similar to a high speed environment where the network speeds are too high to do in-depth packet processing at line rates.

3.3.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) were originally aimed at modelling the complex interactions of the neurons in a biological brain abstractly through a mathematical representation of neurons in a graph structure [Hopfield, 1982]. ANNs are typically applied to classification problems, where the designer presents the system with an input X containing N features $X_0, X_1, ...X_N$. Each feature is passed to a series of input nodes in the ANN. The nodes apply a function and send the outputs to the next layer in the ANN. Each layer transforms the inputs until they reach the output layer. The output layer classifies the inputs [Aleksander and Morton, 1995]. Self Organizing Maps (SOM)[Kohonen, 2001] and Multi-Layer Perceptrons (MLP) [de Sá, 2001] are two popular techniques applied to anomaly/misuse detection using ANNs.

SOMs are an unsupervised learning technique that map input vectors to low dimensional graphs [Polla et al., 2009]. Each neuron in the ANN weights the components of the incoming vector and outputs the vector as coordinates in the low dimensional, typically a 2 dimensional, graph. A neighbourhood function classifies each output vector, according to a set of training inputs processed previously by the ANN.

SOMs have been utilized in a number of domains, especially due to their ability to represent data with high dimensionality [Polla et al., 2009]. Rhodes et al. [2000] used a SOM to classify traffic as either normal or abnormal, training the ANN using vectors derived from network traffic. In order to derive the vectors they recombine packets into application layer streams, instead of using individual packets. Each application stream is monitored by a SOM trained to recognize abnormalities in that specific type of traffic. Portnoy et al. [2001] also used SOMs to cluster network traffic, and chose to extract features from not only the individual connection (duration, protocol type, number of bytes transferred and the flag indicating the normal or error status of the connection) but also from domain knowledge and features extracted from monitoring the connection over time (including failed login attempts, percent of packets with the *SYN* flags, etc.). The vectors consisted of forty-one features over continuous values. Zanero and Savaresi [2004a] rank the SOM algorithm as the best performing out of several they tested for anomaly detection on TCP/IP traffic, highlighting its ability to classify attacks that are difficult to write for misuse detection systems as well as its resistance to polymorphic attacks. SOMs have also been used in a number of other intrusion detection research efforts [Ramadas et al., 2003; Cortada Bonjoch et al., 2002; Lichodzijewski et al., 2002; Kohonen, 2001; Labib and Vemuri, 2002; Girardin, 1999; Patole et al., 2010; Yang et al., 2010].

A Multi-Layer Perceptron is an extension of the original linear perceptron, that approximates non linear approximation functions using multiple layers of nodes. MLPs are often used in supervised learning techniques. Inputs traverse the network of nodes, where each edge applies a weight to the previous output and each node performs a non linear activation function on its input, either activating or deactivating the signal leaving the current node. Each node is connected to every node in the next layer [de Sá, 2001].

MLPs have been used extensively in anomaly detection. Vatisekhovich [2009] and Golovko et al. [2010] proposed an intrusion detection system based on a combination of Artificial Immune System (AIS) and Artificial Neural Networks. The ANN portion of the proposal includes a Multi-Layer Perceptron. MLPs have been used in a number of Intrusion Detection Systems, either as the primary classification method [Ali et al., 2010; Mohamed and Mohamed, 2010] or as part of a hybrid system [Sheikhan and Jadidi, 2009; Pinzon et al., 2010].

3.3.3 Support Vector Machines

Support Vector Machines (SVMs) use nonlinear mappings to transform data into higher dimensions. The algorithm defines a linear optimal separating hyperplane for the higher dimensional data, resulting in a separation of the data into distinct sets. There have been several efforts to use SVMs for intrusion detection, either independently, extending them or combining them with other learning algorithms or feature reduction schemes. Mukkamala and Sung [2003] identify support vector machines as superior to traditional ANNs in Intrusion Detection Systems [Mukkamala and Sung, 2003; Sung and Mukkamala, 2003]. They claim that SVMs scale better, result in higher classification accuracy, and run an order of magnitude faster. Hypersphere-based Anomaly Detection is an anomaly detection technique based on one-class support vector machines [Gornitz et al., 2009]. Hypersphere-based anomaly detection seeks to enclose normal data into a minimal enclosing hypersphere, and any data points that are not found inside the hypersphere are considered anomalous. The hypersphere consists of vectors of data derived from network packet payloads. Data is therefore classified either as normal (contained in the hypersphere) or abnormal (not contained in the hypersphere). Pinzon et al. [2010] combine a SVM with a traditional neural network to reliably classify SQL queries, in their attempts to identify SQL injection attacks. They compare the results of several classification algorithms by testing the algorithm with 705 previously-classified samples and show that the combination of an SVM and a MLP classifies more accurately than several other algorithms, including: Bayesian Networks, naïve Bayes, AdaBoost M1, Bagging, DecisionStump, J48, JRIP, LMT, Logistic, LogitBoost, MultiBoosting AdaBoost, OneR, SMO and Stacking. Teng et al. [2010] implement a fuzzy SVM that addresses the traditional SVMs sensitivity to outliers and noise. Sun and Wang [2010] outline an approach called weighted support vector clustering (SVC), which seeks to improve the performance of SVM in clustering network intrusions and reducing false positives. In order to convert the SVM to a weighted SVC inputs are assigned weights enabling them to contribute differently to learning of cluster boundaries. They show by applying the weighted SVC host algorithm, they obtain better performance than both SVC and K-means. Wang et al. [2009] attempt to improve traditional SVM by introducing particle swarm optimization (PSO) to identify free parameters for the SVM and additionally use binary PSO to obtain the optimum feature subset for intrusion detection. They show that PSO improves the detection capability of traditional SVM. Another examples of SVM applied to intrusion detection is [Yuan et al., 2010].

3.3.4 K-means

The k-means clustering algorithm has been applied to aspects of intrusion detection where addressing smaller subsets of data is less processor intensive than tackling the whole data set at once. For example, Faraoun and Boukelif [2007] use k-means to effectively reduce the sample size forwarded to their neural network classifier to decrease computational intensity, by only sending representative samples from specific categories. While other efforts (e.g [Chairunnisa et al., 2009]) have evaluated k-means as a method to classify data into different types of attacks, its performance as an independent classifier have not been as good as some other popular methods. The unsupervised nature and simple implementation of k-means makes it attractive as a first stage filter and a base line to compare other algorithm performance. Zanero and Savaresi [2004b] also used k-means as one of several clustering algorithms in their research. They found that it did not perform as well as Kohonen's Self Organizing Maps and principal direction partitioning also trained in their research.

3.3.5 Artificial Immune System

Artificial Immune Systems (AIS) attempt to map the functions of the biological immune systems to solve computational challenges such as anomaly detection. Depending on the resolution of the researcher's approach, some element of the computer network, whether it be individual machines, subnets, or even the Internet service provider is treated like a biological entity. The goal of the artificial immune system is to maintain the health of the entity by identifying and suppressing antigens. The approach attempts to model biological functions, and should improve as researchers in biology learn more about how immune systems are able to distinguish between good and bad in biological hosts.

Greensmith et al. [2010] have applied artificial immune system techniques to the detection of port scans. One such technique is the application of the dendritic cell algorithm. In the human immune system, dendritic cells observe environmental features and locality markers, and either activate or suppress immune responses. Essentially, dendritic cells act as anomaly detectors. Given that dendritic cells are capable of performing anomaly detection with high true positive and low false positive rates, Greensmith et al. [2010] postulate that an artificial dendritic cell should also hold those characteristics when applied to anomaly detection in computer networks. They identified four biological signals: pathogen associated molecular patterns (PAMP), necrotic signals, apoptotic signals and pro-inflammatory cytoklins, and mapped them to a set of abstract signals: PAMP, danger signals, safe signals and inflammation. The PAMP functions as a signature of a likely anomaly. High levels of the danger signal indicate potential anomalies. High levels of the safe signal indicate normal functioning. Lastly, there is an inflammation signal that multiplies the other input signals by some factor. Within the system there exists a population of immature dendritic cells. As the dendritic cells are exposed to various signals they become either mature or semi-mature cells. Semi-mature cells result from antigens collected under normal conditions, while mature cells result from possibly anomalous conditions. In the scan detection context the PAMP signal is generated from the rate of ICMP destination unreachable errors received per second. The number of packets sent per second generates the danger signal. The safe signal is represented by the inverse of the rate of change of the packets. Finally, the antigen is determined by systems calls, identified by the PID of the calling process. They show that they can identify scans using this method.

Twycross [2007] introduced the TLR algorithm based on the Toll-like receptor (TLR) family of pattern recognition receptors and applied TLR to intrusion detection, comparing it to implementations of other pattern recognition receptor algorithms twocell and pad1 within the libtissue framework. Libtissue is a software system designed specifically for implementing and evaluating AIS algorithms. Libtissue provides mechanisms to represent tissue compartments. A multi-agent system of cells, antigen and signal interact within the compartments to realize the AIS. Twycross applied the TLR algorithm to identify malicious FTP attacks in a set of otherwise normal FTP transactions. Given that the biological processes involved are complex, explaining each one presented by Twycross is beyond the scope of this literature review. For the sake of brevity I will consider only the twocell algorithm. Twocell is a simple algorithm with type 1 and type 2 cells. Type 1 cells have a cytokine receptor and a antigen receptor, and can produce antigen. They mimic the antigen and signal processing in biological immune systems. The cytokine receptor accept signal from tissues while also accepting antigens from its locality. Type 2 cells contain a cell receptor, a variable region receptor, and a response producer. Type 2 cells are responsible for emulating T-Cells capable of cellular binding, antigen matching, and antigen response. Through the interaction of the two cells binding, the type 1 cell can produce antigen and bind to the type 2 cell, which can then match with the type 1 cell and produce a response. Twycross and Aickelin [2006] use system calls and CPU usage an antigen signal in a twocell implementation to identify anomalies.

3.3.6 Online Machine Learning

So far the literature review has described a number of machine learning algorithms and how they have been applied to anomaly detection. Here I will discuss a series of online machine learning algorithms applied to computer vision. Real time network monitoring shares some of the same characteristics as computer vision. In both you have a stream of information, as opposed to a discrete set of static data points. Equally, in both contexts the data points of interest are surrounded by noise and it is important to discern multiple sources of noise from interesting features. Both share a dynamic environment requiring flexible learning where training is continual, enabling the system to incrementally correct itself over time. Online machine learning algorithms are designed to be quick enough to handle incoming data at varying rates and with low memory requirements, since samples don't have to be stored in memory. However, not all machine learning algorithms are suitable for online processing.

Saffari et al. [2009] proposed a novel approach to visual tracking using an Online Random Forest machine learning algorithm. It effectively combines concepts from both online bagging and extremely randomized forests. They describe a mechanism to build trees and over time trees are replaced to enable the algorithm to adapt to the data stream. Choosing what trees to discard is dependent on the out-of-bag-error of the various trees in the forest. Random Forests are typically resilient to the loss of a single tree, and Saffari et al. [2009] show that the risk of discarding a tree is out-weighted by the adaptability gained by growing new trees. Further research in visual tracking by Saffari et al. [2010] introduced another novel algorithm called Online Multi-Class Linear Programming Boost. While online boosting typically solves binary tasks, Saffari et al. [2010] seek to overcome that limitation. They use a linear combination of weak learners, such as Random Forests. Their technique uses alternating primal-dual descent-ascent across a set of weak learners for calculating weights and age off older samples with newer samples. Interestingly, in their work they also convert the offline multi-class boosting algorithm described in Zou et al. [2008] to an online multi-class boosting algorithm for testing purposes and term it Online Multi-Class Gradiant Boost.

Bordes et al. [2007] demonstrated a successful Online Multi-class support vector machine algorithm named Linear LaRank. Linear LaRank relies on randomized exploration inspired by the perceptron algorithm. The advantage of LaRank is that few passes, or even a single pass over the data can generate error rates that approximate the the final solution. This is an ideal feature for an online machine learning algorithm dealing with a stream of data. SVMs in general were discussed earlier in the literature review, see Section 3.3.3. Saffari et al. [2010] also provide source code for their implementation which include an implementation of Linear LaRank.

3.4 Multi-Agent Systems

This research can fundamentally be characterized as a set of competing multi-agent systems. At the core, there is the Multi-Agent Malicious Behaviour Detection system deployed to protect the network. In opposition to it, there are potentially multiple collections of individual and interacting malicious software agents (malicious multi-agent systems). This section discusses general research in multi-agent systems, focussed primarily on intrusion detection. Intrusion and anomaly detection are dynamic complex problems suitable for Multi-Agent System (MAS) approaches. MASs, both biological and artificial, are ubiquitous. Individual agents in any specific MAS can have varying levels of sophistication. Some agents are aware, interact with, and build trust models of other agents in their environment. Less sophisticated agents may interact with the world as though it is a black box, accepting input and producing output with little or no direct observation of other agents interacting with the same encompassing system. In a MAS, multiple agents are tasked to solve problems with varying levels of interaction and/or communication with other agents and their environments often leading to a partition of the task in terms of type or amount of work, such as Brooks [1991b]; Arkin [1998]; Balch and Arkin [1995]; Nilsson [1984]. For example, in [Wegner, 2003], multiple homogeneous agents were deployed to perform Urban Search and Rescue. There are many multi-agent systems designed to play robotic soccer [Kuhlmann et al., 2006; Baltes et al., 2009; Baltes and Anderson, 2007; Anderson et al., 2002b,a], perform intrusion detection [Abraham et al., 2007; Dasgupta et al., 2005; Álvaro Herrero et al., 2009; Herrero and Corhado, 2009; Rehak et al., 2008; Onashoga et al., 2009], search disaster areas [Wiebe and Anderson, 2009; Wegner and Anderson, 2006; Anderson et al., 2003], perform robotic localization and mapping [Bagot et al., 2008; Bandini et al., 2004], participate in electronic commerce [Dong et al., 2004; Guttman and Maes, 1998; Ketter et al., 2009], and perform in a number of other environments as well [de Denus et al., 2009; Allen, 2009; Parunak, 1997].

Here I will discuss some basic background on agents before describing some existing MASs used in computer security.

3.4.1 Agents

All agents respond in some fashion to the world in which they are situated [Russel and Norvig, 1995]. The goal of agent design is to have the agent respond to its environment in a meaningful way. The agent must have some way of perceiving elements of the environment and interacting with those elements in a purposeful manner. How agents should respond to their environment is highly domain-specific - what is good in one setting might be poor in another. Because domains vary greatly in sophistication, so do the needs of successful agents for those domains. This variation has led to a wide variety of approaches to agent design. Most agent design approaches can be broadly grouped into one of three categories: planning agents, reactive agents, and hybrid agents.

3.4.1.1 Planning Agents

The discipline of artificial intelligence has focused for many years attempting to build intelligence using various forms of representation (e.g. logic, rules) to construct and reason about a model of the world an agent inhabits [Brooks, 1991b]. Central to many classical planning agents is the physical symbol system hypothesis [Newell and Simon, 1976], which states that intelligence operates on a system of symbols, where the symbols represent entities in the environment. Agents typically execute a sequence of three phases; sense, plan and act. In the first phase, sense, the agent uses its sensors to gather data about the world around it. The data is used to update the agent's internal world model, so that changes in the environment are reflected in the symbolic representation maintained by the agent. Following the sensing phase, the agent plans a course of action. The agent analyzes the new internal world model and using some planning algorithm decides what sequence of actions will result in the agent achieving its current goal. Once a sequence of actions has been chosen, the execution phase commences. The execution phase takes as input the sequence of actions that the agent would like to execute and attempts to perform them [Arkin, 1998].

Symbolic reasoning has the advantage of enabling the agent to reason about its past and form elaborate plans for its future. Assuming the sense, plan and act cycle repeats frequently enough, the agent can adjust the plan as things in the environment change. An example of a planning agent is described in [Nilsson, 1984].

Several disadvantages for planning agents have been identified. One major disadvantage of planning agents lies in the difficulty of maintaining accurate internal world models [Brooks, 1991a]. The accuracy of a world model is affected by the frequency that the world model is updated. Because the environments for which agents are designed are often very complex, very elaborate world models are required, and maintaining consistency between such complex world models and the real world is a daunting task [Agre and Chapman, 1991; Brooks, 1990; Chapman, 1989]. If the world model is not updated frequently enough, the world model will fall out of sync with the real world [Brooks, 1990]. If the planning algorithm is applied to a world model that is out of date, the plan produced by the planning algorithm may not be applicable to the real world, since the real world may have changed since the last time the world model was updated [Brooks, 1990; Mataric, 1997].

Another disadvantage is related to the reliance of many planning agents on the physical symbol system hypothesis. If the symbols representing entities in the world model are not sufficiently grounded to physical objects in the real world, they become meaningless [Coradeschi and Saffiotti, 2000; Harnad, 1990]. Additionally, the symbols are task dependent: different tasks require specific representations, making a planning agent only capable of solving the problem it was designed for [Brooks, 1990].

Planning is also very time consuming. Since the world being modelled is complex, the planning system has to be able to predict an exponential number of outcomes. This requires searching through a potentially exponential search space. Heuristics are useful for trimming the search space, but heuristics often sacrifice accuracy [Chapman, 1989]. In any case, planning takes a significant amount of time. Agents in most real world environments do not have significant time to ponder their actions, since rapid changes in the environment quickly make plans obsolete. Taking too much time to plan also highlights another disadvantage: obsolete plans require rerunning the planning algorithm to produce another plan, which may also be obsolete by the time it is ready to execute. Once a plan is prepared to execute, the agent will attempt to complete plan execution until either the plan will obviously fail or a better course of action arises. This requires a level of planning above the planning algorithm to determine when a plan is no longer sensible [Agre and Chapman, 1991]. The plan-replan cycle can lead to agents spending all their time planning, and never executing any actions in the environment. The sense, plan and act cycle in very complex systems inevitably ends up taking too long to allow the agent to interact with a sophisticated world in real-time.

3.4.1.2 Reactive Agents

Reactive agents address some of the downfalls of planning agents. Reactive agents react to the environment without searching or requiring world models [Mataric, 1997]. In general, reactive systems are based on stimulus-response relationships. Stimuli in the environment trigger an immediate response from the agent. Reactive agents are more resilient to noisy data collected from sensors, since there is no world model and therefore noise in collected data is not cumulative. Planning is also eliminated, since responses occur immediately in response to the stimuli requiring very little time [Mataric, 1997; Brooks, 1990]. Eliminating planning makes interactions between reactive agents and their environment much faster than planning agents. Reactive agents are therefore more effective at interacting with the environment in real-time. Reactive systems are based on the assumption that the world is its own best model, since it is always up to date and always contains every detail there is to know [Brooks, 1990]. Reactive agents also address the difficulties with the symbol system hypothesis using the *physical grounding hypothesis* [Brooks, 1990]. The physical grounding hypothesis is based on having a system composed of modules, where each module produces behaviour and the combination of those modules produce more complex emergent behaviour [Brooks, 1990; Mataric, 1997; Arkin, 1998]. Purely reactive agents are robust. Since every module of the reactive system produces responses based on stimuli, if stimuli does not occur in the environment, the associated responses are simply not elicited, having little effect on the emergent behaviour of the agent.

Central to many reactive approaches is the notion of a *behaviour*. Arkin [1998] defines a behaviour as a stimulus/response pair for a given environmental setting that is modulated by attention and determined by intention. Reactive approaches that are behaviour-based take the concept of a behaviour and add structuring to it. Behaviours are organized into larger packages that interact with each other in some way, and often some form of layering is added to mediate between competing behaviours and packages of behaviours

One of the earliest behaviour-based approaches was the subsumption architecture, proposed by Brooks [1986] in response to the limitations of planning agents. Like many reactive systems, subsumption discourages the use of internal world models, allowing only very small pieces of state information to be stored in the behaviour modules. Many agents have been implemented using the subsumption architecture (e.g. [Horswill, 1993; Mataric, 1992; Brooks, 1986, 1989, 1990; Brooks et al., 1999]).

A disadvantage to the purely reactive agents proposed by Brooks is the difficulty of representing complex behaviours without any internal representation and very little memory. Schematically, even simple behaviours such as foraging require complex subsumption diagrams. Since the complexity of the subsumption diagram grows rapidly as the emergent behaviours require more complexity, the scalability of the subsumption architecture is questionable. Another disadvantage of a subsumption architecture is the likelihood of the system becoming trapped in a *local minima*. A *local minima* is a position where every action that the agent executes will lead it back to the same position. Since the responses to a sequence of events can potentially repeat endlessly, the agent can become trapped by continuing to repeat the same actions over and over again and making no progress. Getting trapped in local minima highlights the need for agents that are able to use past experience to modify future behaviour.

Schema-based agents emerged as an approach to agent control that realizes the advantages of purely reactive agents while loosening some of the constraints proposed by Brooks [Arkin, 1998]. Schema-based agents retain the reaction speed achieved by purely reactive agents by discouraging complex world models and instead containing multiple simple world models that are efficient to maintain. This addresses many of the disadvantage of purely reactive agents. Like many approaches to robot control, schema-based agents support the notion that there is a lot more to be gained by increasing the functionality of reactive approaches before adding planning facilities.

Schema-based agents are based on schema theory. Briefly, schema are a mapping of perceptions to actions associated with the perceptions. They contain the information necessary to encode agent behaviour by means of the sensory data required to illicit an action, where the computational process for how the action is performed is embedded in the schema [Arkin, 1998].

Many schema-based agents have been implemented (e.g. [Arkin, 1992; Cameron et al., 1993; Balch and Arkin, 1995]). While schema-based agents are meant to address the disadvantages of purely reactive systems, some disadvantages still remain. Schema-based agents are not proactive. They lack the ability to reason about the future or the past having little to no representation of the world. In addition, schemas are intended to be mapped onto the hardware for which they are designed, making it difficult to move systems from one platform to another [Arkin, 1998].

3.4.1.3 Hybrid Approaches

There are a number of control architectures that attempt to combine the advantages of both symbolic reasoning and reactive control [Graves and Volz, 1995; Arkin and Balch, 1997; Gat, 1992; Lyons and Hendriks, 1995; Lee et al., 1994]. The majority of these systems use a reactive control system as their base. The agent reacts to the environment as the environment changes and in general is designed to handle the agent's short term, underlying behaviour. Often the reactive component is responsible for very time sensitive tasks that could pose a risk to the agent. Above the reactive subsystem, there is generally a higher level planner. The planner is responsible for planning ahead and directing the high level goals of the agent. Ideally the planner can take as much time as it needs to maintain a world model and plan using that world model, since while planning, the agent is continuing to function using the reactive subsystem. The agent does not have to wait until the planner is ready to send actions before the agent can perform productive work.

Knowledge used to represent internal world models for the planning component of the hybrid system can make the behavioural configurations of the underlying reactive system more versatile by adding knowledge about the task and the environment. Additionally the dynamic nature of the internal world model may provide insight into ways of achieving goals that the reactive system is not able to, since it has no representation to work with [Mataric, 1997].

The planning component of a hybrid system still shares the disadvantages of planning agents described in Section 3.4.1.1. If the environment is not stable or consistent from one time step to the next, the planner may still develop useless plans [Arkin, 1998]. Continually re-planning will result in a slowdown of the overall system, especially if the plans are not relevant. The issue of physical symbol grounding is still present, even with a reactive subsystem: if the symbols used by the planner are not well grounded they will be of little use [Brooks, 1990]. Finally, the world model may still be inaccurate, adding to poor plans [Mataric, 1997].

3.4.2 Multi-Agent Systems in Computer Security

Having described the basic concepts surrounding intelligent agents, we can now examine putting these together into intelligent multi-agent systems. As mentioned previously, there are many domains to which this can be oriented, and for the purposes of this thesis I will be focusing on work in computer security that adopts a multi-agent approach.

There are several existing multi-agent intrusion detection systems such as CIDS [Dasgupta et al., 2005], APHIDS [Deeter et al., 2004], AAFID [Spafford and Zamboni, 2000] and SCIDS Abraham et al. [2007]. The focus of these intrusion detection systems is on the distribution of the detection, where the detection system is a MAS composed of multiple detection agents that communicate. This is distinct from the research at the core of this thesis, which is a system for the detection of malicious multi-agent systems, i.e. where the intruders themselves are potentially interacting agents.

CIDS [Dasgupta et al., 2005] is a multi-agent intrusion detection system based on Cougar, an open source agent framework. The agents in CIDS are partitioned into one of 4 types: manager agents, monitor agents, decision agents and action agents. Security nodes encompass one of each agent, while a security node society is a group of security nodes. Each agent within a security node has a specific task: manage the communications and commands between agents, monitor the host or network collecting data and identifying possible intrusions, make decisions based on data and analysis provided by the monitor agents, or take some form of action as recommended by the decision agent. The security nodes are distributed among the subnets of a network. The agents themselves use standard artificial intelligence approaches such as those discussed in section 3.3.1.

Deeter et al. [2004] designed APHIDS, a mobile agent-based programmable hybrid intrusion detection system. They identify the principle challenges for intrusion detection as false positives as well as management and correlation of large amounts of data. The proposed system consists of a set of distributed agents deployed throughout the network on service providing hosts, such as web servers and mail servers, as well as network monitoring devices such as fire walls and intrusion detection systems. The agents are managed through a scripting language which describes trigger and task agents, and effectively pushes the analysis and response to the distributed agents. The distributed nature of the system reduces bandwidth and farms out the resource usage to the hosts. The script language enables new agents to be added as well as supporting future host types by abstracting away from the specific systems to the scripting language understandable by each agent in the distributed system. APHIDS appears to be an intelligent distributed management layer that incorporates the various existing security and service providing hosts.

Spafford and Zamboni [2000] describe a system called AAFID. AAFID is a distributed detection architecture. Spafford highlights the strengths of an agent-based distributed architecture, namely that each agent can operate independently, can be reconfigured, restarted, or even removed entirely without impacting the distributed system. Agents can also be tested independently and introduced into the environment afterwards. They also propose measuring the performance of agent-based architectures based on the following properties: continuous running, fault tolerance, resistance to subversion, minimal overhead, configurability, adaptability and scalability. The AFFID architecture contains a hierarchy of entities, composed of monitors, transceivers, filters and agents. At the lowest level the agents subscribe to filters and analyze data for signs of intrusion. The filters provide the agents with system independent information to analyze. Each agent is autonomous and searches for instances of some specific interesting event. Any number of agents can exists within a single host. The agents in a particular host report their findings to the hosts transceiver. Each host's transceiver then communicates with a monitor. The transceivers are responsible for aggregating data from all of the agents in a host as well as controlling the agents, sending them commands, configurations and start and stop instructions.

A monitor is responsible for communicating with one or more host transceivers. Monitors can correlate alerts from several transceivers and produce reports on activities. Monitors can also be organized in a hierarchical fashion, with higher order monitors communicating with lower level monitors. Notice that even though the agents are distributed the system as a whole still has a hierarchical form, resulting in a central point for reporting higher in the architecture. One important aspect of this architecture is that agents are not capable of communicating with each other: instead all communications must be sent to a transceiver.

Rehak et al. [2008] introduce the concept of trust into the analysis of Netflow data. In their model, agents are responsible for analyzing traffic flow data to determine if a specific flow is malicious. The system then aggregates the results of individual agent analysis to produce a trust score based on reputation. More specifically, their approach has three layers. The first layer consists of traffic acquisition and preprocessing. Data is collected using hardware-accelerated NetFlow probes. Next, the data is passed to a cooperative threat detection layer, consisting of specialized agents capable of detecting some malicious anomaly. The agents use an extended trust model, and collectively decide the degree of a flow maliciousness using a reputation mechanism. Finally, an operator and analyst interface layer provides network operators with a mechanism to interpret the output of the collective agents. The strength of the system is that is reduces the cognitive load of the operator by reducing false positives produced by agents whose detection mechanism are prone to error. The trust model only passes on those events where the agent collective has determined that the flow is likely malicious.

Abraham et al. [2007] use co-operative intelligent agents distributed across a network to validate their fuzzy rule-based classifiers, intended to detect intrusions in a network. Like similar work in distributed intrusion detection using agents, the agents are located at various hosts, and each on is responsible for implementing some part of the fuzzy rule system. They propose both a light weight and a heavy weight SCIDS. The heavy weight SCIDS operates against 41 variables, similar to many intrusion detection systems before it (see Section 3.7), while the light weight operates on 12 variables resulting from feature selection on the 41 features using decision trees. They implement decision trees, linear genetic programs, and fuzzy classifiers for intrusion detection. This research highlights the importance of controlling agents interacting with other agents in a hierarchical manner, such that information flows between agents and up to agents higher in the hierarchy, flowing information to the administrator.

3.5 Plan Recognition

All malware characterized earlier (Section 3.4.2) has an intended end to the behaviour in which it engages. This is true whether the given malware is an isolated entity or a set of collaborating entities. In artificial intelligence, the subfield interested in reconstructing an agent's intentions from its observable behaviour is known as *plan recognition* Kautz and Allen [1986]. This problem is much more complex when involves a set of observed agents rather than one, since each may have its own intentions distinct from that of the group [Banerjee et al., 2010]. At the same time, however, the actions of one agent in a known system may allow hints at what other agents may be intending [Intille and Bobick, 1999]. Little work has been done directly on treating distributed malware infections as malicious MASs, and studying the behaviour of those systems. However, work in plan recognition has been performed in other areas, and shows promise for application to security. In competitive multi-agent environments, one team of agents attempts to model the behaviours of another team of agents in order to gain some advantage as they compete for resources. Plan recognition is important for determining the intentions of individuals and teams of autonomous agents.

PHATT [Geib and Goldman, 2009] is an algorithm for plan recognition that uses a Bayesian approach. Geib and Goldman [2009] seek to improve plan recognition by reducing the number of simplifying assumptions that previous systems required to be effective, such as restricting the number of plans observed agents pursue, requiring plans to be ordered, etc. PHATT makes use of plan libraries organized to represent plans as partially ordered AND/OR trees, where each node is augmented with prior probabilities, specifically prior probabilities of root goals, method choice probabilities, and probabilities for picking elements from the pending sets. The tree allows for method decomposition, choice points and ordering constraints. Using Bayes' rule, PHATT observes an agent performing a set of actions and then determines where those actions fit in a preset plan library, as part of one of several sequences of actions to achieve some goal or as an action in a pending set for some goal. Note that PHATT does not attempt to deal with deceitful agents, and assumes that the actions that agents execute are all intended to help achieve their goals. Geib and Goldman [2009] provide a formal grammar for defining a plan library and explain how to apply algorithms to plan trees built from the proposed grammar.

Kuhlmann et al. [2006] present an autonomous system, UT Austin Villa, capable of observing and classifying behaviours of multi-agent systems. In general the system uses knowledge gleaned from past observations to inform a multi-agent team of strategies to use against an opponent multi-agent team. UT Austin Villa is capable of observing groups of agents playing robotic soccer, and by using predefined formation patterns and a set of statistical features at specific times throughout the game, it classifies the opponent teams behaviour. In essence the autonomous coach uses features to classify team behaviour into one of several predefined classes. The methods appear to be specific to robotic soccer as they use a ball attraction model, that would require modification to be applicable to the information security realm. The work does provide inspiration for multi-agent modelling.

Zilberbrand [2009] proposes a hybrid system consisting of a symbolic plan recognition algorithm capable of detecting anomalous behaviour and a utility-based plan recognizer tasked with reasoning about the expected cost of hypotheses. The goal of their system is to overcome previous systems reliance on plan recognition accuracy as the primary measure of performance, as opposed to the utility of the information in the observing agent, as well as overcoming multiple simplifying constraints imposed by previous systems. A number of previous plan recognition systems, when given a set of observations, will return some number of likely plans, ignoring plans that are not likely. However, Zilberbrand [2009] examines the impact of ignoring plans that are not likely, however represent a high amount of risk or opportunity if they represent the agents true intentions, and implemented the utility-based plan recognizer for that purpose. Zilberbrand [2009] also extended standard symbolic plan recognition by adding feature detection trees for dealing with complex multi-faceted observations, added reasoning about time duration in plan execution to rule out hypothesis early in the reasoning process and added extensions to deal with lossy observations and interleaved plans.

3.6 Malware Collection

Malware collection is an important aspect of research in anomaly detection. As the evolution of malware outpaces the efforts of network security practitioners to defend their networks, researchers must constantly develop new techniques that are effective against modern and future malware, as opposed to past examples of malware, whose behaviours may not be encountered in the wild anymore (or pose as much of a threat). Evaluating any system requires realistic test cases, and so to evaluate my work a reliable source of malware was required. Section 6.2 discusses how traffic was gathered for the evaluation of my approach. Here, I review prior work in malware collection for comparison purposes.

Often malware collection is performed using one of two types of honeypots (see Section 2.4.4), low interaction honeypots and high interaction honeypots. Low interaction honeypots are designed to provide an attacker with a subset of functionality, the minimum required to elicit an infection with some interaction between the attacker and the victim. However, it is not intended to be a fully operational machine, and is minimalist in nature, which significantly improves its scalability [Baecher et al., 2006; Provos, 2004]. High interaction honeypots are intended to be much more functional, and are in fact a real, fully featured machine for the attacker to interact with.

It is less scalable, but often more information about the attack is available from the machine [Baecher et al., 2006; Balas and Viecco, 2005].

Gu et al. [2007] use a honeynet in order to test their Bothunter architecture. The honeynet is composed of a Drone Manager, a high interaction-honeynet system and a DNS/DHCP server. The Drone Manager keeps track of a list of available machines. When a connection attempt is made to the target IP range, the Drone Manager either chooses a new virtual machine to forward the connection to, or forwards the connection to a potentially infected virtual machine already assigned the target IP address. As machines become infected, a list of "tainted" machines is built consisting of machine that are under attack. The high interaction-honeynet system is an Intel Xeon 3 GHz, dual core system with 8 GB of memory responsible for providing the virtual machines. The authors found that typically 9 Windows XP instances, 14 Windows 2000 instances and 3 Linux FC3 instances were capable of handling the network requests, given that many of the infection attempts were unsuccessful, allowing the machines to be recycled to accept new connections. The DNS/DHCP server is responsible for providing the virtual machines with IP addresses and answering DNS requests from the hosts. The authors note that most infection attempts did not succeed on unpatched version of Windows 2000 and Windows XP, which might be due to malware able to distinguish true hosts from virtual machines.

Baecher et al. [2006] propose Nepenthes, a platform designed to collect self-replicating malware in the wild, essentially a honeypot (see Section 2.4.4). It focuses on emulating only the vulnerable portions a service/machine to increase efficiency. The architecture is formed by a number of modules built on top of the nepenthes core, which is responsible for the network interface. The modules pass the malware through several stages, starting with vulnerability modules designed to mimic only the relevant parts of exploitable services. When a virtual service is exploited, the payloads are passed to a shellcode parser. If the shellcode is successfully parsed and it contains instructions to download malware, the location is passed to a fetch module capable of using a variety of protocols to obtain the malware. Depending on the objective of the user, several different modules can perform actions on the fetched malware. Nepenthes also has reverse shell emulation and virtual file systems for infection vectors that require them. Additionally, Nepenthes is designed to identify exploits that diverge from expected behaviour and redirect the exploit to a high-interaction honeypot with the goal of capturing zero-day attacks (for more information on zero-day see Section 2.2.1). Note that Nepenthes is only capable of collecting autonomously spreading malware and, like many honeypots, has difficulty capturing hit list attacks. The authors claim that one instance of Nepenthes can emulate 16000 IP addresses, emulating 2000 concurrent honeypots. In testing, Nepenthes observed 5.5 million exploitation attempts in 33 hours, and collected 1.5 million binaries, 508 of which were unique.

Provos et al. [2008] use a web-honeynet to verify drive-by downloads. The web-honeynet is composed of a large number of Microsoft Windows images in virtual machines. In order to check if a URL is malicious a given virtual machine will load a fresh Microsoft Windows image with an unpatched version of Internet Explorer. The browser is directed to the URL, and a variety of heuristic checks and anti-virus scans are performed on the virtual machine to see if it was compromised. Each

HTTP response is scanned with a variety of anti-virus products. Additionally, the system is monitored for abnormal activity such as newly created processes, changes to system registry and file system changes. The abnormal activities are combined into a heuristic score representing the likelihood that a URL was malicious. On average out of the one million URLs that the web-honeynet could process daily, about 25,000 of those were classified as malicious.

Potemkin [Vrable et al., 2005] is a honeyfarm that addresses the trade-off between scalability, fidelity and containment. Vrable et al. [2005] propose taking advantage of idle memory and cpu cycles by exploiting several infection characteristics, primarily that given the large IP space of virtual machines (VM), only a very small number of IP addresses are active at any given time and therefore do not require the resources of an instantiated virtual machine. Their implementation involves a gateway machine that filters packets into their honeyfarm, dropping scan packets and packets that are not destined to the target IP range. It also contains the honeyfarm by applying similar filters to outgoing packets. A Virtual Machine Manager within the honeyfarm is responsible for instantiating reference images, which are snapshots of machines already booted with their applications loaded and ready to accept connections. As packets enter the network through the gateway destined for a new VM, a clone is spawned that adjusts its IP address and then picks up the network packets. The authors term this *flash cloning*. Another mechanism to aid with scalability is delta visualization. In order to save on resources, each spawned clone maps all of its code and data pages from the reference image, instead of creating a copy of them. The reference map of code and data pages is not writable and therefore maintains a pristine status. There is copy-on-write functionality that allows machines to deviate from the reference image.

Wang et al. [2006] introduce HoneyMonkeys, a set of virtual machines that host monkey programs in a variety of different operating systems. Each monkey program drives an unpatched browser and attempts to mimic human browsing habits. The goal is to infect the virtual machine by following malicious links, if they exist. The authors use an application called Strider Tracer [Wang et al., 2003] to monitor file changes to the virtual machines indicative of infection. In a one month trial of HoneyMonkeys, Wang et al. [2003] discovered 752 unique malicious URLs with the capability to exploit unpatched versions of Microsoft Windows XP. The malicious URLs were hosted on 288 web sites. Further trials identified 741 web sites hosting 1,780 exploit URLs. In their trials they deployed 10 browsers per virtual machine, one virtual machine per physical machine, and a total of 20 physical machines. The Windows machines had varying patch levels applied to them, with fully patched Windows machines used to verify if a malicious URL identified earlier on unpatched machines was actually a zero-day attack (see Section 2.2.1). Wang et al. [2006] demonstrated that they could identify zero day attacks before the rest of the security community using their system.

For more examples of researchers using virtual machines for honeypots see [Anagnostakis et al., 2007; Moshchuk et al., 2006].

One issue identified by Stone-Gross et al. [2009] in using virtual machines in malware analysis is that malware can do system checks to identify when a machine is hosted virtually, by performing checks on the virtual hardware serial and model numbers. These numbers are often consistent across virtual machines.

Wang et al. [2010] explore the detection of honeybots and honeynets. They propose that a botmaster can use ethical and legal bounds applied to security researchers to verify whether or not a particular bot in their botnet is actually a honeypot. Typically security professionals are liable for the traffic originating from their honeypots, therefore they must filter the traffic leaving their honeypots to ensure they do not damage public property. The botmaster instructs a bot to send out malicious traffic to another bot (sensor) under the botmaster's control. The botmaster then verifies with the sensor bot to see if the malicious traffic was received. If the malicious traffic was not received, the botmaster assumes that the originating bot is either a honeypot, or in a secure network. In either case, the original bot can be discarded. Examples of malicious traffic are attempts to infect other machines, low-rate port scanning, and email spam, all of which should be blocked by the security professionals managing the botnet to avoid legal proceedings. Wang et al. [2010] also illustrate that implementing DNS sinkholes are detectable by botmasters using master DNS controller lists that can be routinely compared to a bot sensor's own DNS resolutions. Lastly, they demonstrate that if a botnet's controller is hijacked, the same mechanism that detects honeypots using malicious traffic can equally be applied to identifying hijacked botnet controllers.

Wang et al. [2010] also address how a P2P botnet might detect honeypots. They designed a two-stage reconnaissance worm, the two stages are spearhead and main-force. The spearhead infects a machine, and then attempts to infect some other number of machines. Once spearhead manages to infect some fixed number of other machines, it downloads the main-force stage. Main-force contains the payload required to formally join the P2P botnet, which may consist of an authorization key. The authors are taking advantage of the fact that security researchers will not allow their honeypots to deliberately infect the number of machines required to obtain main-force. Some of the effect of two-stage reconnaissance worms can be mitigated by deploying a honeynet, as opposed to a honeypot, and having honeypots infect other honeypots without sending malicious traffic past the gateway. However, Wang et al. show that further modifications to the two-stage reconnaissance worm can detect such activity. The trade off is really in how much the botmaster intends to invest in ensuring the botnet can detect honeypots.

Wang et al. [2006] identified three methods that their HoneyMonkeys could potentially be identified by malware, preventing the malware from executing. First, adversaries could discover the IP addresses known to host HoneyMonkeys or honeynets in general, and refuse to infect those machines. Second, the malicious website could try to detect honey clients by attempting to interact with the user, through dialog boxes or CAPTCHA tests for example. Third, there are a number of methods for an application to determine if it is in a virtual machine, for example: executing instructions with high virtualization overhead and detecting anomalies in the elapsed time, or identifying the execution of x86 opcodes typically reserved for specific virtual machines.

3.7 Feature Extraction

A network flow or the memory space of a target machine can have an abundance of features. While some of those features are important for identifying malicious and anomalous behaviour, many of those features can be ignored. A substantial amount of research focuses on identifying important features that provide maximum information gain. As such, feature selection is important in intrusion detection, as a good set of features can improve detection accuracy and reduce processing time. Large feature sets can impact the running time of anomaly detection algorithms. While a minimal set of features can enable real-time processing by reducing the execution time of the anomaly detection algorithms. In particular, for ANN, the training increases with the square of number of features in each input vector [Kauzoglu, 1999].

Gonzalez [2009] studied feature selection by applying several feature selection algorithms to a set of 248 features for network intrusion detection. Gonzales applied the following algorithms: accuracy rate with best first search, Decision Tree (C.45), accuracy rate with a genetic algorithm, RELIEF-F, Probability of Error and Average Correlation Coefficient (POEACC), Generalized Relevance Learning Vector Quantization Improved (GRLVQI), median Bhattacharyya and minimum surface Bhattacharyya methods. Gonzales concludes that there are a subset of key features required for generality: ports, packet size, timing attributes, and other protocol-specific indicators. Specifically, RELIEF-F produces a feature subset that maintains generality across all tested classifiers using 16 features from the original 248 for Transmission Control Protocol flows, for example client port, server port, time since last connection, and maximum data sent. The other feature selection algorithms selected subsets between 12 to 27 features and were often able to improve the classification accuracy of at least one of the 4 classification algorithms. See Table 3.1 for an example subset of features. See [Gonzalez, 2009] for a complete list of the 248 features.

Feature Subset		
client port	server port	med data control a b
max data wire	max data ip	mean data control a b
max data control	req sack b a	q3 data control a b
max segm size a b	max segm size b a	max data wire b a
min segm size a b	avg segm size a b	max data ip b a
max data wire a b	max data ip a b	var data control b a
Time since last connection		

Table 3.1: Sample subset of features chosen by Gonzalez [2009]'s algorithm.

Lorenzo-Fonseca et al. [2009] addressed the issue of reducing the number of characteristics necessary for presenting an input to an ANN with minimal loss of specificity. A TCP/IP traffic input consists of over 400 potential features, that Lorenzo-Fonesca et al proposed to reduce to 20 principal components using a principal component methods, and claimed they could express over 95% of the original input samples total information. Using a Multi-Layer Perceptron ANN: composed of 20 neurons in the input layer, 36 neurons in the hidden layer and 1 output layer, they classified traffic as either Normal or Attacks with 99.41% accuracy on their own test data and a detection rate of 90% on the DARPA intrusion detection evaluation data. Not only does the research demonstrate promising results for ANN in general for the IDS task, it also demonstrates the increase in efficiency attainable by reducing the inputs layer from a potential 400 inputs to 20.

Zanero and Savaresi [2004a] use approximately 30 input entries for each TCP/IP packet they process in their anomaly detection engine. They also highlight the requirement to search for anomalies within individual packets (intra-packet correlation) as well as anomalies that emerge from groups of associated packets (inter-packet correlation) [Zanero and Savaresi, 2004a]. However, later in their paper they identify that the choice of features were done through trial and error, and later experiments used fewer features and gleamed better results for a system that analyzed groups of 10-20 packets at a time. They indicate that proper feature selection requires more research.

The DARPA data set, which has been a standard for testing anomaly detection in many anomaly detection research efforts since 1998, describes network traffic using 41 features for each TCP/IP connection. The data set is intended to simulate a typical U.S. Air Force LAN. However, the simulated LAN was subjected to a large number of network attacks. A set of 24 attack types were introduced into the data set that fall under the following four categories: Denial of Service, Remote to User, User to Root, and Probing. The features include such things as protocol, flags, duration, destination host count, is host login, is guest login, etc. For a full list of features see [Sheikhan and Shabani, 2009].

DARPA features	Sheikham	Chebrolua	Mukkamala	Khor
duration			X	
protocol_type	X			
service	X	X	X	F1, F2, F3, F4, F5
src_bytes		X	X	F2, F4
dst_bytes		X	X	F1, F2, F3, F4, F5
logged_in	X	X		F1, F2, F3, F4, F5
num_compromised	X			
root_shell				F1, F4, F5
count		X	X	F1, F2, F3, F4, F5
srv_count	X	X	X	
serror_rate		X		
rerror_rate	X			
srv_rerror_rate	X	X		
same_srv_rate	X			
diff_srv_rate	X			
srv_diff_host_rate		X		F1, F4
dst_host_count		X	X	F1, F4, F5
dst_host_srv_count	X	X	X	F2, F4
dst_host_same_srv_rate	X			
dst_host_diff_srv_rate	X	X		F2, F4
dst_host_srv_diff_host_rate	X			F1, F4
dst_host_rerror_rate	X			F2, F4, F5
dst_host_srv_rerror_rate	X			

Table 3.2: Features identified by several algorithms as important to anomaly detection.

Sheikham et al. [2010] test the effect of reducing the feature set of the original KDD 99 on classification rates and learning time for neural attack recognizers. They take 41 features described in the KDD 99 data and through preprocessing, scale and map them to linearly scaled values (e.g. in the range of [0,1]) and integer ranges (e.g. 1,2,3...N). They rank the features by applying logistic regression on the Chi-squared feature values, effectively ranking the features by evaluating subsets of feature from size 1 to 41. They found that using 15 features in a MLP they could achieve better overall performance than several other learning algorithms using varying feature set sizes. For a list of the 15 top ranked features see Table 3.2.

Khor et al. [2009] constructed BN Classifiers to evaluate the performance of their feature selection algorithm. The feature selection proposed using a correlation based feature selection subset evaluator and a consistency subset evaluator to form two feature subsets (F1 and F2). They formed a third subset (F3) as the intersection of the first two subsets (F3 = F1 intersect F2), and a fourth subset (F4) as the union of the first two subsets (F4 = F1 union F2). They also allowed domain experts to add features they deemed important to the third subset to create a fifth subset (F5 = F3 + domain knowledge). For a list of the individual features in each feature set see [Khor et al., 2009]. They conclude that classification accuracy can be maintained with the reduction of the feature set. Important features can be identified by combining multiple feature selection algorithms, and taking the intersection of the selected features, while the resulting feature set can be improved using additional features recommended by domain experts. The features of the set F1, F2, F3, F4 and F5 are listed in Table 3.2

Botminer [Gu et al., 2008a] uses the following features to identify command and control communications: time, duration, soure IP, source port, destination IP, destination port, number of bytes transferred in both directions and number of packets transferred in both directions.

Mukkamala and Sung [2003] use a performance base ranking method (PBRM) to determine what features are important for identifying anomalies in network traffic. They also used a unique ranking method, Support Vector Decision Function Ranking Method (SVDFRM) for choosing the most important features for anomaly detection using SVMs specifically. Both methods begin with the 41 features of the DARPA data set and iteratively transform the feature set into three subsets of important features, secondary features and unimportant features. For a list of important features see Table 3.2 and [Mukkamala and Sung, 2003].

Chebrolua et al. [2005] applied Bayesian networks (BN) and Classification and Regression Trees (CART) as well as a combination of the two, to feature selection in the design of an IDS. Using BN they seek to identify a Markov blanket defined as follows: the Markov blanket of a feature T, $MB(T)$ of a BN. The set of parents, children, and parents of children of T. $MB(T)$ is the minimal set of features conditioned on which all other features are independent of T, i.e. for any feature set S $P(T|MB(T),S) = P(T|MB(T))$. The BN MB model for feature selection identified 17 important features. For a full list of 17 features see [Chebrolua et al., 2005].

Chebrolua et al. [2005] CART algorithm is based on binary recursive partitioning with nodes representing items to be classified and each node splits on different features. Features that contribute the most to classifying inputs at each split increase in importance. The CART algorithm yielded 12 important features. The 12 important features are listed in Table 3.2 and found in Chebrolua et al. [2005].

Kloft et al. [2008] propose a learning algorithm to automatically identify features for anomaly detection. Their approach is based on an extension to support vector data description (SVDD) using one-class anomaly detection. They show that it is feasible to select optimal feature combinations automatically with machine learning and demonstrated their technique on HTTP traffic.

Provos et al. [2008] use features such as "out of place" IFRAMEs, obfuscated JavaScript and IFRAMEs to known distribution sites as features in web-pages to help identify attacks that redirect browsers to malware distribution websites.

3.8 Impact of Encryption and the VPN

Encryption and Virtual Private Networks add an additional layer of complexity to anomaly detection [Goh et al., 2010, 2009]. Assuming traffic is passed as plaintext, devices designed to inspect the contents of network packets can perform matching against the content of the packet using several well known string matching algorithms. Sessionizing, the act of constructing sessions from individual packets sharing the same IP addresses and ports, is time consuming, as the state of the traffic flow must be kept in memory. If packets are encrypted, the device must either decode the packets on the fly, which can delay in-line processing, or match against the encrypted content. Given that the encrypted content of any given packet is unlikely to match a content-based rule, encrypted malicious traffic is more likely to circumvent an IDS. Decoding the content of the packets requires the IDS to have access to all of the keys to perform decryption. Depending on the encryption used, a full session of data may have to be collected before decryption can occur.

Encryption on the wire increases the value of host-based intrusion detection, given that once traffic arrives on the host it is decoded to be used by the application responsible for those communications. The host-based sensor may be able to delay matching against the traffic until the application responsible for generating the traffic performs decryption.

Virtual Private Networks (VPN) introduce similar difficulties. However, placement of network sensors can alleviate many of the difficulties. Since VPN traffic consists of a tunnel, the IDS should monitor the traffic after it has been reassembled from the tunnel instead of matching against the VPN traffic encapsulating the true network traffic.

Encrypted traffic is quickly becoming ubiquitous, and in general all encrypted traffic on a network should be documented by system administrators. There is some merit to blocking all unknown encrypted traffic entering or leaving a network, but given that legitimate encrypted traffic is popular, system administrators are responsible for learning what encrypted traffic must be allowed and what can continue being blocked. Gu et al. [2007] identify encryption as a challenge for the effectiveness of their detection architecture.

3.9 Summary

In this chapter I described existing research in botnet detection, and reviewed a range of work in machine learning as applied to anomaly detection. I reviewed the artificial intelligence concepts of agents and plan recognition, and then introduced multi-agent systems and their applications in computer security. I discussed techniques for both collecting malware and extracting features from collected traffic. Finally I touched on the impact encryption has on anomaly and intrusion detection. The next chapter is concerned with describing the architecture of the *Multi-Agent Malicious Behaviour Detection*.

Chapter 4

Multi-Agent Malicious Behaviour Detection Architecture

4.1 Overview

The major contribution of this thesis is a framework for Multi-Agent Malicious Behaviour Detection, which is described in this chapter. This framework aids network defenders in the detection, mitigation, and study of malicious multi-agent system through interaction with both the human network defender and the malicious multi-agent system. The architecture of the Multi-Agent Malicious Behaviour Detection system is described in seven sections. First, I define and discuss the concept of a Multi-Agent Malicious Behaviour Detection system. Second, I summarize the distinction between the architecture described in this chapter and both standard misuse detection and anomaly detection. Third, I described the role of the network defender, and the teleautonomous aspects of Multi-Agent Malicious Behaviour Detection. Fourth, I describe the Multi-Agent Malicious Behaviour Detection Agents that make up the system and the function they serve. Namely, I will describe the Traffic Source Agent, Feature Source Agent, Machine Learning Agent, Alert Source Agent, Protocol Analysis Agent, Observer Agent, and Traffic Manipulation Agent. Fifth, I illustrate a set of agents deployed in a fictional network to demonstrate how the agents in the system interact. Sixth, I describe the communication infrastructure for the agents in the system. Finally, I give a detailed example of how sessions of traffic are processed by a hypothetical Multi-Agent Malicious Behaviour Detection system.

4.2 Multi-Agent Malicious Behaviour Detection

As described in Section 1.1, Multi-Agent Malicious Behaviour Detection is inspired by the hypothesis that malicious software agents continue to become more intelligent, autonomous and cooperative and techniques from artificial intelligence, namely machine learning and multi-agent systems, are valuable in detecting, discovering and mitigating malicious multi-agent systems. Framing network attacks as potentially intelligent malicious multi-agent systems benefits both the computer security and multi-agent system domains. Such framing demands novel techniques for learning behaviour from intelligent malicious multi-agent systems (Section 1.5.4), where those systems are themselves designed to evade detection wile carrying out a series of tasks. The key to Multi-Agent Malicious Behaviour Detection is the generation of features derived from interactions between malicious software agents. These features can then be generalized across a variety of current and future malicious multi-agent system instances, and exploited to detect them. A feedback mechanism encourages network defenders to extend the Multi-Agent Malicious Behaviour Detection system's capability with human expertise. This methodology relies on both machine learning and manual techniques to infer behaviour patterns from malicious multi-agent systems. Care is taken to ensure that the behaviours derived from the malicious code are not so general that they are similar to other legitimate software agents present and operating on the network, which malicious software agents may try to disguise themselves as. However, given that there are several legitimate groups of software agents behaving in similar ways, the architecture includes a method for whitelisting specific applications where possible.

Malicious software agents in the wild infect machines in a variety of ways (see Section 2.2.2). Once a malicious software agent establishes a presence on a victim machine, several key behaviours are commonly presented in order to achieve the malicious multi-agent system's intended goals (Section 1.5.4). First, malicious software agents often beacon to other instances in a malicious multi-agent system within the same local network, organization or autonomous system. Equally, malicious software agents often beacon out to malicious multi-agent system controllers responsible for managing the various malicious software agents, and may initiate an update behaviour. Second, in order to communicate, malicious software agents have methods for identifying other instances and those methods may mimic common traffic on the network. For example a domain name look up, a connection to a predefined location, or the form a broadcast takes may all serve to identify similar malicious software agents. Third, malicious software agents typically have a means for propagation, such as copying malicious code into emails, or embedding scripts into HTTP requests (Section 1.5.4). Malicious software agents may also attempt to escalate their privileges on the local network through a variety of attacks, often targeting local mail servers, domain name servers, or authentication servers such as the domain controller in a Microsoft Windows based network, to facilitate propagation.

The malicious behaviours addressed by the Multi-Agent Malicious Behaviour Detection Architecture, presented in Section 1.5.4, are those that can be identified on the network using a variety of packet capture and feature extraction methods. The architecture is intended for detection capability in real-time. The architecture is also capable of promoting the propagation of a malicious multi-agent system in a controlled manner, in order to increase the framework's opportunity to learn from a malicious multi-agent system's communications.

4.3 Distinction

Here I discuss the distinction between a misuse detection architecture, an anomaly detection architecture and the Multi-Agent Malicious Behaviour Detection Architecture. The first distinction is the focus. Anomaly detection and misuse detection are techniques for identifying malware, while Multi-Agent Malicious Behaviour Detection focuses on detecting sophisticated malicious multi-agent systems. All three approaches are capable of detecting both individual malicious software agents as well as malicious multi-agent systems. However, there are subtle differences between these.

Whereas misuse detection (see Section 2.5.1) is typically reactive to very specific instances of malicious activity, and only effective against the specific case a given rule was defined to identify, Multi-Agent Malicious Behaviour Detection generalizes from previous malicious multi-agent system communications and focuses in on those behaviours that are likely to be repeated across future varied malicious multi-agent system instances. Often misuse detection depends on string matching for terms previously identified or for known malicious domain names and IP addresses. These predefined terms are likely to change over time, as attackers acquire new IP addresses or change their domain names, or obfuscate their code, rendering the rules useless. However, some behaviour must remain consistent in spite of such changes, and the aim of Multi-Agent Malicious Behaviour Detection is in part to identify those characteristics.

Anomaly detection (see Section 2.5.2) attempts to identify the unknown malicious behaviour through a variety of classification methods. Ideally, all traffic is classified as either normal or abnormal. Unfortunately, anomaly detection requires a substantial sample of normal traffic, and as the anomaly detection system is applied to a variety of different networks, the concept of normality can vary wildly. What is normal on one network is abnormal on another, and vice versa. One key aspect of anomaly detection is that it finds anomalies, not specifically malicious behaviour. Starting from a base set of knowledge (often some training set of network traffic), everything different beyond some tolerance level from the training set is termed anomalous and must be investigated. Every new legitimate network behaviour will also trigger the anomaly detection engine and may produce a number of false positives. Additionally, there is the risk that existing malware on a network will be included as normal traffic since it is active when traffic samples are taken for the training set of normal traffic.

Existing malware that is covertly communicating within the network could potentially become whitelisted. In general, attempting to define normality makes very little sense on the Internet as a whole. Multi-Agent Malicious Behaviour Detection differs from standard anomaly detection in that it starts with learned seed behaviours that are then generalized and added to a set of behaviours that identify malicious activity (as opposed to merely anomalous). Additionally, Multi-Agent Malicious Behaviour Detection does not consider the problem to be that of pure classification, but rather a study of intelligent malicious multi-agent systems capable of blending in with normal traffic.

Misuse detection is a reactive approach, which focuses on finding instances of known malicious behaviour, while anomaly detection is a proactive approach seeking to identify novel instances of malicious activity. Multi-Agent Malicious Behaviour Detection is somewhere between the two: it seeks to find novel attacks based on lessons learned from previous attacks. Additionally, both misuse detection and anomaly detection are effective when applied against simple attack scenarios, such as scan detection, where the behaviour is well defined. However, as machines become more powerful, increased processing capability provides more available resources that in turn allow malicious software to operate unnoticed by taking advantage of idle cycles. This in turn allows for more sophisticated and intelligent malicious software agents, which are better able to avoid traditional anomaly detection.

Now that I have given a high level description of the Multi-Agent Malicious Behaviour Detection system and how it is distinct from both misuse detection and anomaly detection, I will briefly discuss the role of the network defender with respect to Multi-Agent Malicious Behaviour Detection.

4.4 Network Defenders

In Section 1.1 I introduced the concept of network defenders as human individuals responsible for protecting computer networks. In computer security, network defenders are also referred to as IT security specialists, security professionals, white hats, system administrators, etc. For Multi-Agent Malicious Behaviour Detection, the network defender is the human in the loop, providing expertise, feedback and exerting control over the Multi-Agent Malicious Behaviour Detection Agents. As discussed earlier (Chapter 1), detecting malicious multi-agent systems is a complex problem. While a completely autonomous system for protecting computer networks is ideal, some network behaviour detection tasks remain easier for trained human experts to recognize and validate than the current state of the art in artificial intelligence can support. Multi-Agent Malicious Behaviour Detection therefore seeks to be a *teleautonomous* system - one which allows its automated components a level of autonomy, but also allows the inclusion of input from a human. The Multi-Agent Malicious Behaviour Detection framework engages the network defender through an intuitive user interface (Section 4.5.7), and takes advantage of the defender's expertise to verify the system's hypotheses, as well as enabling network defenders to provide feedback into the system. Multi-Agent Malicious Behaviour Detection aims to reduce the cognitive load of the network defender while still providing situational awareness through the Observer Agents and Protocol Analysis Agents (Section 4.5.5). Taking advantage of teleautonomy enables Multi-Agent Malicious Behaviour Detection to take advantage of human-level intelligence while still pushing the boundaries of artificial intelligence. This research reinforces the blending of human and machine intelligence and provides a research platform to explore human and artificial intelligence interaction. Next, I present the individual Multi-Agent Malicious Behaviour Detection Agent types that make up the Multi-Agent Malicious Behaviour Detection system.

4.5 Multi-Agent Malicious Behaviour Detection Agents

The Multi-Agent Malicious Behaviour Detection Architecture consists of a number of cooperating Multi-Agent Malicious Behaviour Detection Agents. Multi-Agent Malicious Behaviour Detection Agents are categorized by one of seven roles: traffic source, features source, machine learning, alert source, protocol analysis, observation, and traffic manipulation. Figure 4.1 illustrates the various Multi-Agent Malicious Behaviour Detection Agent roles, and gives an example of communication between the various agents. The green boxes represent a network packet. In Figure 4.1, a Traffic Source Agent (the main means by which the system interacts with incoming network streams, Section 4.5.1) copies the packet into memory, and initiates cooperative processing of the packet by the other Multi-Agent Malicious Behaviour Detection Agents. At the bottom left, a Traffic Manipulation Agent inserts a replacement packet, represented by a red box, in front of the original packet back into the network. The replacement packet represents the potential response of the Multi-agent Malicious Behaviour Detection Framework to the given traffic. Though the system may have additional side effects (e.g. through the actions of warned network defenders), the replacement packet allows my framework to directly affect the network. For example, a replacement packet intended to get to the destination host before the original packet (Section 4.5.6), as in a spoofing attack [Yan et al., 2006]. Such a replacement packet could be a modified DNS packet to redirect the host to a benign server, or an HTTP response with a malicious link removed, or even a TCP packet with the FIN flag set in an attempt to close down the connection. For the sake of simplicity only one Multi-Agent Malicious Behaviour Detection Agent per role is illustrated in Figure 4.1. However, the system is intended to allow multiple Multi-Agent Malicious Behaviour Detection Agents for each role. This is further elaborated in the sections that follow, along with a more sophisticated example once all agent types are described. I will refer back to Figure 4.1 while describing each Multi-Agent Malicious Behaviour Detection Agent role.

The Multi-Agent Malicious Behaviour Detection Architecture is intended to be dynamic: agents occupying the roles described in this section can be added or removed as the system does its work. Regardless of the individual agent role, each Multi-Agent Malicious Behaviour Detection Agent must first announce its presence to the other agents currently in the system by way of a broadcast message. The message includes a unique identifier for the new agent and an indicator of what role it intends to fill. Existing Multi-Agent Malicious Behaviour Detection Agents in the system respond to the broadcast indicating their unique identifier and role, allowing agents to establish directed communication between on another.

The ability for agents to join or leave the system in this way increases the system's

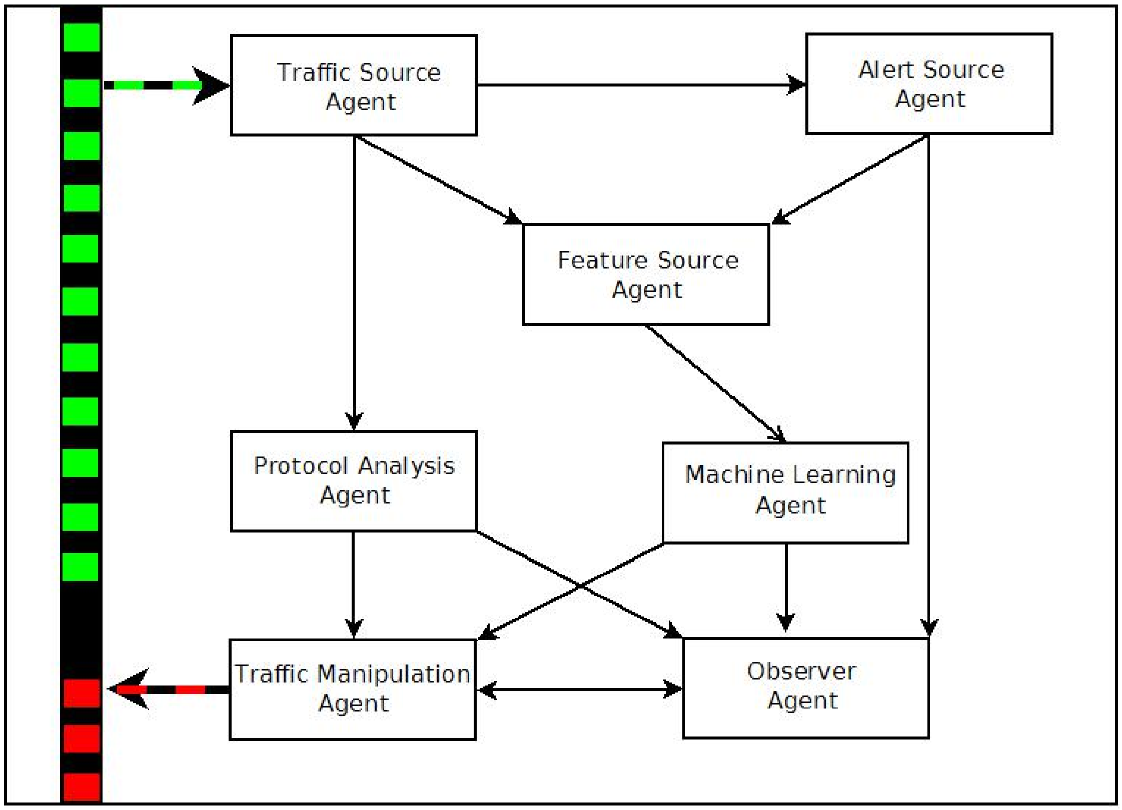


Figure 4.1: The seven agent roles and an example of information flowing through the system.

overall resilience in four ways. First, it improves the system's ability to cover complex network topologies. For example, in a segmented network certain segments may be covered by a variety of Traffic Source Agents (Section 4.5.1). Each Traffic Source Agent can examine network traffic independently, providing different forms of abstracted perception (i.e. information of interest) to other agents in the system based on their needs. Second, it enables resource management. Multi-Agent Malicious Behaviour Detection Agents can be instantiated on a number of different machines in the protected network. However, each machine may have other processing priorities. When processing resources are limited, Multi-Agent Malicious Behaviour Detection

Agents can leave the Multi-Agent Malicious Behaviour Detection system in order to give those resources back to the host machine. The Multi-Agent Malicious Behaviour Detection system can still carry on, minus the abilities by that particular agent. Third, multiple Multi-Agent Malicious Behaviour Detection Agents provide redundancy. If one Multi-Agent Malicious Behaviour Detection Agent crashes, or even becomes compromised, other agents can be instantiated to dynamically take its place. Finally, a network defender could potentially design and implement a new type of agent and deploy it into the existing system and dynamically remove the agent if it does not perform as expected.

The following sections describe in detail the individual Multi-Agent Malicious Behaviour Detection Agent types. A detailed example of how a population of Multi-Agent Malicious Behaviour Detection Agents cooperate is provided in Section 5.4.2.

4.5.1 Traffic Source Agents

Traffic Source Agents provide other Multi-Agent Malicious Behaviour Detection Agents in the system access to some traffic resource. Traffic Source Agents are passive listeners, taking advantage of network taps, ethernet cards in promiscuous mode or previously collected traffic. Each Traffic Source Agent is capable of processing network traffic to a degree required for Multi-Agent Malicious Behaviour Detection Agents in the system to exploit it. Traffic Source Agents provide access to the traffic source in as close to the raw data packets as possible, while providing some structure to simplify further processing. The experimental networks used for the implementation of this architecture (Chapter 5) are IPv4 over ethernet, requiring each Traffic Source

Agent implementation to provide Multi-Agent Malicious Behaviour Detection Agents with a method to subscribe to IP layer traffic. Generally though, a Traffic Source Agent should not be limited to IPv4 or even IP.

Placement of Traffic Source Agents in the network topology is important, but not as critical as typical network sensor architectures. As discussed in Section 2.3.1, typically sensors should exist at choke points throughout the network, and these may impact network performance, such as in the case of in-line detection. The novelty of Traffic Source Agents is that multiple Traffic Source Agents can exist throughout the network, each sharing network traffic from its own view of the environment, providing Multi-Agent Malicious Behaviour Detection Agents in the system the opportunity to pick and choose the traffic sources in which they are interested.

Each Traffic Source Agent is a source of shared perception in the Multi-Agent Malicious Behaviour Detection system. Figure 4.1 shows an example Traffic Source Agent reading network packets from some network medium and distributing the resulting structured network packets to a Feature Source Agent, a Protocol Analysis Agent, and an Alert Source Agent. The structured network packets will have important fields, such as IPs, ports, and flags parsed with pointers into the raw data, making it easier for other agents to work with them. Having one agent do this parsing work removes the need for all the other agents having to parse their elements of interest individually, and speeds the system by removing redundant work. In general, any Multi-Agent Malicious Behaviour Detection Agent can subscribe to a Traffic Source Agent: Figure 4.1 is just an illustrative example.

The challenging aspect of a Traffic Source Agent's role is ensuring that it is capable of keeping up with the speed of packets that are being sent across the network medium. It is not uncommon for home networks to be connected to the Internet at speeds of up to 40-60 Mbits per second, while personal home networks commonly contain routers capable of achieving 100 to 1000 Mbits per second. Raw network capture libraries, such as tcpdump, are capable of capturing packets at fairly high rates. However, the major requirement of Traffic Source Agents is providing an interface for higher level programming languages to manipulate low level packet structures enabling efficient analysis and the capability to insert modified packets back into the network fast enough to allow the Multi-Agent Malicious Behaviour Detection system to interact with malicious multi-agent systems. If Traffic Source Agents cannot keep up with the packets, then the whole Multi-Agent Malicious Behaviour Detection system will either fall behind in processing or be forced to drop packets. Programming for speed is the essential challenge in implementing a Traffic Source Agent (Section 5.5.1). As network speeds increase, specialized hardware is required to keep up with packets. This research focused on using standard hardware that was readily available and inexpensive. Reliance on open source libraries was necessary to produce high level Multi-Agent Malicious Behaviour Detection Agents, but also limited the capture speed attainable for this research. Section 5.5.1 discusses more about the specific libraries used. A deep understanding of networking was required to try and get the most out of the packet capture libraries to improve the Traffic Source Agent's performance.

By providing multiple Traffic Source Agents and abstracting away the details of the traffic medium, the system has the potential to blend in a number of traffic sources to expose the system to artificial attack traffic as well as the live network traffic. For example, a Machine Learning Agent (Section 4.5.3) might be trained while no malicious network traffic is available by playing back previously-recorded traffic.

The implementations of two traffic sources, *PcapFileTrafficSource* and *PcapLive-TrafficSource*, are further described in Section 5.5.1.

4.5.2 Feature Source Agents

Feature Source Agents are similar to Traffic Source Agents in that they provide Multi-Agent Malicious Behaviour Detection Agents with a source of shared perception. However, there is an important distinction between traffic sources and feature sources. Feature sources process the raw packets into a set of features - an intermediate step - so that other Multi-Agent Malicious Behaviour Detection Agents can consume those features instead of the raw data. Feature sources provide a further layer of abstraction from the raw data. While the feature source is not concerned with the mechanism for capturing packets, it must understand what a packet is, and what features are important. Features can be derived from single packets, or groups of packets. Feature Source Agents are closely associated with both Protocol Analysis Agents and Traffic Source Agents (see Section 4.5.1 and 4.5.5). The output of a Feature Source Agent is a *feature set*, which contains no network packets, just the relevant abstracted features such as: the number of packets processed, the average size of the application layers in the packets, and the time since the machines involved last connected. Feature Source Agents contain the logic to perform this feature extraction

(Section 3.7).

A significant challenge for Feature Source Agent design is identifying the number of packets sufficient for deriving relevant features to send to other Multi-Agent Malicious Behaviour Detection Agents. If too few packets are used to derive a feature set, then the set will not be useful in identifying valuable features. However, if the Feature Source Agent waits for too many packets, then the delay between the Multi-Agent Malicious Behaviour Detection system observing the packets and the feature set reaching all of the interested Multi-Agent Malicious Behaviour Detection Agents reduces the overall value and reaction time achievable by the Multi-Agent Malicious Behaviour Detection system. To deal with the range of packets one might wish to use as a basis for deriving a feature set, three modes for the Feature Source Agent are provided: deriving features from single packets, a complete session, or a fixed number of packets.

Consider deriving features from single packets. This technique provides the fastest reaction time for the Multi-Agent Malicious Behaviour Detection Agents. If a single packet is sufficient in identifying the features of malicious multi-agent system agent communications, then Multi-Agent Malicious Behaviour Detection Agents can be notified of the packet features with fairly low latency. However, using only a single packet reduces the number of features that can be derived. Recognizing important features such as the amount of data exchanged between two IP addresses requires tracking the bytes for the duration of the communication between two IP addresses. Many such features identified in previous research (Section 3.7) require multiple packets. Features derived from a single packet out of context might not provide enough information for the Multi-Agent Malicious Behaviour Detection Agents using these features to make informed decisions. For example, in the case of a three way handshake during a TCP connection set up, a SYN packet is not likely to contain features worthy of forwarding on to other Multi-Agent Malicious Behaviour Detection Agents. However, in some circumstances, such as standard DNS responses and requests, a single packet contains all of the relevant features.

Next, consider deriving features from entire sessions. A session can typically be defined by five primary features: the server and client IP addresses, the server and client port numbers, and the protocol. Identifying packets that belong to a session is straightforward: a hashing structure that derives keys from the five primary features can be used to index a set of features, one for each session. As packets are identified that belong to a specific session, the session features are updated with the relevant parts of the new packet. Identifying when sessions are complete, and then publishing the feature sets to other Multi-Agent Malicious Behaviour Detection Agents is critical to efficient operation. The problem appears to be simple in the case of a TCP session. Each TCP session should begin with a TCP handshake involving a SYN, SYN/ACK and ACK. Then each TCP session should end when either both ends of the session sending packets with a FIN flag set, or one of the machines sends a packet with a RST flag set. However, in reality a large number of TCP sessions do not end properly. In fact, quite often there is no proper TCP session close. Instead, the TCP session eventually times out. The way the Feature Source Agent deals with improperly closed TCP sessions will impact its capability to react to events in the system. If the Feature Source Agent sees a FIN flag set on a packet from both the client and server ends of the session, then a feature set is published and the session can be removed from the indexing structure. If a proper close is not observed, then the session remains in memory. Sessions that remain in memory too long impede the system's capability to respond to threats in the network. A timeout must be chosen that increases the likelihood that TCP sessions with a proper tear-down are not truncated, and minimizes the delay for timed out TCP sessions. The issue is compounded by connectionless transport protocols such as UDP, where relevant features can be derived across a complete session, but there is no obvious mechanism for the Feature Source Agent to identify the end of the session, without perhaps understanding the overarching application protocol. Additionally, being connectionless, a UDP session is misleading. For the purposes of this thesis, two hosts observed exchanging UDP datagrams will be considered by convention, as participating in a session, with the host that initiates the exchange considered the client. A host that responds, after receiving an initial UDP datagram, by sending back a UDP datagram will be regarded as a server and its corresponding UDP srouce port is regarded as open. Depending on the number of sessions tracked at any given time, the available memory of the Feature Source Agent becomes a factor. Too many sessions in memory at one time could tie up large amounts of system memory and effectively render the Feature Source Agent useless, as in a denial of service attack like those described in Section 1.5.4.

Finally, one could estimate a fixed number of packets that provide a high likelihood of capturing complete sessions while minimizing the number of truncated sessions and delay from deriving features to notifying additional agents. However, estimating a fixed number of packets does not prevent very short sessions from having to timeout. Suppose, after studying the average session length on a network, it is determined that most sessions complete after ten packets. Any sessions less than ten packets long, for example DNS, would require an additional timeout since they will not reach the ten packet threshold. Instead, average session lengths would have to be determined on a per protocol basis, requiring additional computations to recognize the variety of protocols on a network. Even within a given protocol, the actual number of packets can vary wildly. For example, consider HTTP. There are a number of different data types and lengths that are transferred across HTTP that make it very difficult to determine an appropriate number of packets.

The problem of estimating a useful number of packets can be handled by dividing sessions into categories, where the number of categories depends on the implementation (Section 5.5.2). In this research, a set of categories represent traffic types that fit into one of the 3 modes described above. However, other undefined modes may prove more efficient for types of traffic not considered in my research. In any case, the choice of mode has to be made early in the session processing, and simple heuristics are preferred. Choosing a mode based on the port number, protocol, or some other simple characteristic of the first or second packet is an efficient mechanism. While not perfect, a simple heuristic reduces the potential delay between receiving the packets to a session and the features of that session being published for other Multi-Agent Malicious Behaviour Detection agents. See Section 5.5.2 for more information on how Multi-Agent Malicious Behaviour Detection categorizes sessions.

This leads to the next challenge associated with deriving features from sessions, determining what features are sufficient to identify the target malicious multi-agent system behaviours (Section 1.5.4) and providing those features to other Multi-Agent Malicious Behaviour Detection Agents. I have previously reviewed much work on feature extraction (Section 3.7). For the purposes of this work, the primary features extracted from TCP and UDP sessions are based on those described in by Gonzalez [2009] and presented in Table 3.1. However, special cases are made for HTTP and DNS, given that the literature reviewed identifies them as rich sources of malicious software agent communication (see Sections 3.2, 3.6 and 3.7). The features in Table 3.1 are applicable to the HTTP, TCP, and UDP sessions. However, only a subset of the features are applicable to the DNS sessions.

DNS traffic is unique in that each session consists of a very small number of packets and it is used frequently in malicious multi-agent system communications (Section 3.2). Each DNS packet is also rich with features that can be quickly extracted into feature sets. Table 4.1 contains a list of the additional features extracted from DNS packets.

DNS Features	
Identifier	Created by the program to represent the query, used to match requests to replies.
Flags	16 bits reserved for various flag bits, including the aa, tc, rd, ra and z bits.
Question Count	The number of DNS queries in the DNS request.
Answer Count	The number of answers in the DNS response.
Name Server Count	The number of name servers referenced in the DNS response.
Additional Record Count	The number of additional records in the DNS packet.
Host Name Features	Various features for ASCII host name strings found in the DNS packet.

Table 4.1: Features extracted from DNS packets.

Additional features are also extracted for HTTP traffic. The features are based on the HTTP request and response headers found in sessions on port 80 and 8080.

When a Feature Source Agent identifies an HTTP method, or an HTTP response, the Feature Source Agent extracts various HTTP header field values, such as User-Agent, Via and Referrer. For a full set of extracted header fields see Table 4.2.

HTTP Features	
User-Agent	The user agent string of the user agent.
Via	A list of proxies that a request has passed through.
Referrer	Indicates a web-page that redirected to the current web-page.
Host	The domain name of the server.
Cookie	A previously set cookie.
Server	A name for the server in an HTTP response.
Location	The content location in an HTTP response.
Set-Cookie	Sets a session cookie.
URL	The universal resource locator.

Table 4.2: Features extracted from HTTP sessions.

The Multi-Agent Malicious Behaviour Detection system is designed with extensibility in mind for additional Feature Source Agent types. So while this research introduces five specific Feature Source Agent types (see Section 5.5.2), these are implemented so that new agents (e.g. a SMTP Feature Source Agent) could be added by extending one of the existing types. For more information specifically on the implementation of these agents and how they might be extended to add support for additional features, see Section 5.5.2.

4.5.3 Machine Learning Agents

Machine Learning Agents consume feature sets provided by Feature Source Agents. Section 3.3 introduced a number of machine learning algorithms that have proved successful in identifying malicious behaviour. As discussed in Section 1.6.6, this research does not seek to advance machine learning techniques *per se*, but rather exploits available machine learning work to further computer security, while exploring combinations of techniques which have not been explored previously in this area in an online manner. From the point of view of the computer security researcher, the difficulty of working with machine learning algorithms is understanding what machine learning algorithms are applicable to the texture of data available in the computer security environment and how to adapt information from that environment to existing machine learning algorithms.

With respect to the applicable algorithms, I chose to leverage a series of online machine learning algorithms used for visual object tracking in computer vision described by Saffari et al. [2010]. The algorithms include Online Random Tree, Online Random Forest, Online LaRank, Online Multi-Class Linear Programming Boost, and Multi-Class Gradient Boost. The reasons for choosing these particular algorithms were twofold. First, the types of algorithms matched those that have been used in previous computer security research, as discussed throughout Section 3.3, with the exception that in the computer security work discussed the algorithms were offline. Second, while traditionally machine learning has been used in offline learning to identify network threats, my Multi-Agent Malicious Behaviour Detection Framework aims to introduce online malicious multi-agent system communications analysis to provide network defenders with the capability to react in real-time to malicious software agent behaviours, as described in Section 1.5.4. This novel approach highlights the similarities between requirements for visual object tracking and Multi-Agent Malicious Behaviour Detection. Both tasks require continual learning as the environment changes. In addition, each must process and react quickly enough to input to meet real-time demands. While in the object tracking domain the algorithms are adjusting to a dynamic world, in this research the algorithms are adjusting both to the network defender's feedback and changes in the networking environment. To be clear, there are a number of other potential algorithms that could have been implemented as Machine Learning Agents, and the design of Machine Learning Agents is intended to allow for quick hypothesis testing of any available algorithm with minimal changes by providing a clean interface to wrap the machine learning algorithm in and abstract it from the work performed in both Traffic Source Agents and Feature Source Agents. The provision for several different algorithms contributes to the novelty of this work by ensuring a modular and relatively easy way of exploiting advances in machine learning. The representative sample of machine learning algorithms are integrated into Machine Learning Agents. Through various experiments (Chapter 6) I further demonstrate the capability of Machine Learning Agents to provide value within the Multi-Agent Malicious Behaviour Detection system, making a novel contribution to computer security.

With respect to adapting information from the environment to meet the requirements of machine learning algorithms, the Feature Source Agents, discussed in Section 4.5.2 provide the majority of adaptation. By feeding sets of features represented by a series of floating point values derived from network traffic, the Feature Source Agents enable the clean interface between raw traffic and machine learning input.

Machine Learning Agents operate by associating features pulled from the underlying network traffic with classifications derived from human network defender expertise. In traditional machine learning an algorithm is presented with a training sample set to learn how to classify samples in the problem domain [Alpaydin, 2004].

The training set should consist of a representative sample of data from the problem domain where the classification of each data sample is known. Training is achieved by presenting a data sample to the machine learning algorithm, with its associated classification and the machine learning algorithm updates to compensate for the new data sample and classification. As the training set is exhausted, the machine learning algorithm learns to classify the data. After training, the algorithm is presented with a labelled testing set. In order to test the machine learning algorithm's accuracy, each sample in the test set is passed through the machine learning algorithm and the class assigned to the data sample by the machine learning algorithm is compared to the desired classification provided by the test set. Typically the accuracy of the machine learning algorithm is defined by the number of correct classifications divided by the total number of samples [Alpaydin, 2004]. One or more training and test sets may be employed with the goal of achieving a high degree of classification accuracy on a test data set before deploying the trained algorithm against the real world problem. While the scenario just described has been used previously to classify network traffic [Alpaydin, 2004], it relies on three assumptions. First, that one can derive a set of network traffic that is representative of the expected network traffic. Second, that once a baseline of network traffic is derived, it will remain stable enough to continue to use as a reliable mechanism to train the machine learning agent. Third, the labelling in the test set has been done correctly.

As discussed in Section 3.3.6, online machine learning tackles a different problem. Network traffic is dynamic. Applications are installed and removed, protocols change and malicious software agents evolve. A baseline of traffic is likely to be rapidly outdated. Additionally, providing a reliable set of labelled data for learning is difficult, especially given that malicious multi-agent system communications are attempting to blend in with common protocols. Instead, the Machine Learning Agents are designed to learn over time and remain dynamic as Feature Source Agents publish network features for them to consume. Some feature sets will be labelled by Alert Source Agents, while many feature sets will not. Machine Learning Agents consume the feature sets, training against the labelled feature sets and attempting to classify the unlabelled feature sets. The labelled feature sets provide Machine Learning Agents with an opportunity to adapt to network traffic by taking advantage of previous knowledge, providing meaningful assistance to a network defender tasked to identify novel malicious multi-agent system communications. For more information on how traffic is labelled in the Multi-Agent Malicious Behaviour Detection system, see Section 4.5.4.

Importantly, and further elaborated in Section 4.5.4, there are two strategies investigated here for what sorts of features are ultimately passed on to the Machine Learning Agents and how it impacts their performance. The *uniquely malicious* technique labels only what is perceived to be malicious and effectively tasks the Machine Learning Agents to identify network traffic with similar features as those identified through misuse detection, therefore providing a mechanism to discover new instances of malicious multi-agent system communications that exhibit similar behaviour to known instances. Alternatively, the *classify all* technique, while much more expensive to operate, uses signatures like those used in misuse detection with the addition of signatures designed to classify benign traffic as well. The Machine Learning Agents are then tasked with identifying all traffic it sees as one of several protocols where a subset of those protocols are malicious. In the second mode, signatures designed to identify benign traffic act as a whitelist and should be updated as applications are installed and removed from machines in the protected network. Analyzing the output of the Machine Learning Agents requires that attention be paid to the situation where the Machine Learning Agent identifies something that looks like malicious multi-agent system communications, but also the situation where something is classified as benign, but the next most likely classification is a malicious one. The similarity is quantified by a confidence value for a malicious behaviour class. Even though the confidence value for the malicious behaviour class is less than the confidence for the benign protocol class, the confidence value for the malicious behaviour class still provides evidence for potential malicious behaviour. For example, a malicious software agent beacon might share a number of features with an HTTP post. The malicious software agent mimics the HTTP post so well that the Machine Learning Agent has a 0.75 confidence that the disguised beacon is benign. However, the beacon class confidence is 0.20 and all other confidence values are insignificant. Given that the Multi-Agent Malicious Behaviour Detection system assumes a malicious multi-agent system will attempt to mimic HTTP posts, a 0.20 confidence might warrant further analysis from a network defender. More on such analysis techniques will be presented in Chapter 6.

While significant work was required to wrap existing machine learning algorithm implementations into Machine Learning Agents, Section 4.5.4 explains how the data provided by Alert Source Agents aids in grounding the feature set data to network traffic and labels that are comprehensible to network defenders.

4.5.4 Alert Source Agents

There are a significant number of existing misuse and anomaly detection systems (Section 2.4). A common theme for both misuse and anomaly detection involves using a form of *signature* that triggers an *alert*. An alert flags a potential threat to the system that must be reviewed (by either a human or intelligent system), and is typically identified by a *signature ID*. The alert itself is the combination of network traffic and misuse or anomaly detection signature.

Alert Source Agents provide alerts in a manner that is meaningful to the Multi-Agent Malicious Behaviour Detection system. These can either be done by monitoring an existing detection system and converting that system's alerts, or by generating such alerts themselves. This leaves the opportunity of folding any existing detection system or collection of systems into my framework, and also allows the construction of custom detection systems as part of this. Existing misuse and anomaly detection systems have extensive knowledge bases, consisting of potentially thousands of signatures. Additionally, many such detection systems have well defined and unique mechanisms for network defenders to write new signatures or modify existing ones. In order to incorporate existing detection systems, Alert Source Agents monitor detection systems and convert their alerts to something meaningful to the Multi-Agent Malicious Behaviour Detection system. Incorporating existing systems enhances the Multi-Agent Malicious Behaviour Detection system by: providing network defenders with context; enabling network defenders or other agents to map signature IDs from existing detection systems to hypothesis made by the Multi-Agent Malicious Behaviour Detection system; and providing a feedback mechanism for network defenders and other agents capable of writing or modifying existing signatures for malicious multi-agent system detection.

Existing misuse/anomaly detection systems are often incompatible. Therefore, each Alert Source Agent is written specifically to interpret the source detection system's alert format, and feed a single standardized alert message into the Multi-Agent Malicious Behaviour Detection system using a unique identifier that ties the alert message back to the original network traffic and the particular detection system that generated the alert. Existing Multi-Agent Malicious Behaviour Detection Agents receive alert messages from Alert Source Agents and make decisions based on the alert messages received.

The common mechanism for Multi-Agent Malicious Behaviour Detection Agents to communicate about traffic is through feature sets. By introducing alert messages into the Multi-Agent Malicious Behaviour Detection system, agents can match up feature sets with alert messages and supplement the feature set by adding a signature ID that associates the feature set with known malicious multi-agent system communications or some other traffic type. Alert Source Agents enable the Multi-Agent Malicious Behaviour Detection system to take advantage of existing misuse/anomaly detection knowledge bases, through the signature IDs that are added to feature sets.

External to the Multi-Agent Malicious Behaviour Detection system, misuse/anomaly detection processes generate alerts based on network packets matched against detection signatures. Alert Source Agents, through interacting with the misuse/anomaly detection processes, identify alerts, convert them into alert messages, and pass the alert messages on to agents in the Multi-Agent Malicious Behaviour Detection system. Recall from 4.5.1, Traffic Source Agents intercept packets independent of the misuse/anomaly detection processes and share them with other Multi-Agent Malicious Behaviour Detection Agents. Feature Source Agents (Section 4.5.2) integrate the packets into feature sets. Feature set messages and alert messages enter the system independently. However, a feature set and an alert message may be associated with one or more of the same original network packets. If a feature set generated by a Feature Source Agent and an alert message generated by an Alert Source Agent contain the identical source IP, destination IP, source port, destination port and protocol, Multi-Agent Malicious Behaviour Detection Agents who receive both messages match up the feature set with the alert message and enrich the feature set with the detection signature ID. When Machine Learning Agents receive a feature set enriched with a signature ID, the agent can use the labelled feature set to learn to identify *similar* traffic based on features of the traffic as opposed to the detection technique from the external detection system that originally generated the alert. More on how Machine Learning Agents take advantage of labelled features sets is discussed in Section 4.5.3.

Part of the difficulty with implementing Alert Source Agents is determining how much of the incoming traffic should be labelled with alerts from misuse detection systems in order to provide Machine Learning Agents with enough samples to distinguish between benign traffic and malicious multi-agent system traffic. As discussed in Section 4.5.3, whether the Multi-Agent Malicious Behaviour Detection system is trying to *classify all* traffic or *uniquely malicious* traffic significantly changes the amount of work that Alert Source Agents must perform. When Machine Learning Agents are only expecting labels for malicious multi-agent system traffic, as in uniquely malicious, Alert Source Agents can forward each alert they identify through their internal mechanisms. However, when *classify* all is employed, the Alert Source Agent may only forward a subset of alerts to manage resources.

Coordination between Traffic Source, Alert Source and Feature Source agents is a non-trivial problem. Consider, for example that Traffic Source Agents are attempting to pass traffic up to Feature Source Agents as fast as they can. However, if Feature Source Agents process features faster than Alert Source Agents process traffic/alerts, they will miss incoming alerts and feature sets will pass through the Multi-Agent Malicious Behaviour Detection system onto the Machine Learning Agents unlabelled. The same can be said of Feature Source Agents potentially running slower than Alert Source Agents. In order to compensate, each Feature Source Agent maintains metrics on how many feature sets it labels, and how many alerts it receives that it is unable to match to a feature set resident in memory. If too many feature sets are published without labels, or too many alerts arrive without a matching feature set, Feature Source Agents will keep feature sets cached for a longer time. There is a constant balancing act between the Multi-Agent Malicious Behaviour Detection system's ability to keep up with traffic and react quickly to incoming traffic, and the finite resources available to Feature Source Agents. The coordination is more easily achieved when only malicious traffic alerts are passed through Alert Source Agents.

The advantage to the approach described in this research is that while the Multi-Agent Malicious Behaviour Detection system is likely to fall behind, it can compensate for some data loss. Machine Learning Agents (Section 4.5.3) are not expecting perfectly labelled data. As such, the Multi-Agent Malicious Behaviour Detection system is biased towards dropping alerts in favour of keeping up with the traffic in order to enable real-time reaction capability. More specifics on messaging between the various agents in the Multi-Agent Malicious Behaviour Detection system is provided in Section 4.6.

4.5.5 Protocol Analysis Agents

Feature Source Agents are responsible for scanning traffic and looking for features that other agents will be interested in. However, the features that Feature Source Agents are tasked to identify are low-level and intended for other automated agents. Since human network defenders will be interacting with the Multi-Agent Malicious Behaviour Detection system, they will need information that is more highly abstracted and tuned for their needs. The feature sets generated by Feature Source Agents are of less value in their raw format to a network defender. Protocol Analysis Agents are intended to satisfy the need for information in a format friendly to human network defenders.

Protocol Analysis Agents are designed to provide real-time functionality for network defenders to seek out situational awareness. Protocol Analysis Agents understand a specific application layer protocol from the TCP/IP model and extract contextual information, making it available to network defenders and Multi-Agent Malicious Behaviour Detection Agents alike. As discussed in Section 2.2.1, malicious multi-agent systems are known to exploit and mimic non-malicious protocols to engage in a number of behaviours such as beaconing, propagating, denying, ex-filtrating, and updating (Section 1.5.4). For example, malicious multi-agent system agents can propagate in emails, beacon using DNS or ex-filtrate data over HTTP. In some cases malicious software agents adhere to the protocol standard, and only by understanding the protocol and extracting meaning from the application layer data can the malicious behaviour be detected. In other cases, while malicious software agents make an attempt to disguise their actions by mimicking a known protocol, the resulting communications don't adhere to the protocol standards. For example, a malicious software agent may attempt communications with a malicious multi-agent system using an encrypted tunnel on port 53 to ex-filtrate data, an attempt to mimic DNS traffic. When possible, a network defender can query Protocol Analysis Agents in real-time for contextual information regarding activity on the network.

Consider the following example. An Alert Source Agent monitors a hypothetical misuse detection system. The detection system's knowledge base contains a signature that identifies all DNS queries for the host name *mybad.host.com* as beaconing behaviour for a malicious multi-agent system. The Machine Learning Agents, having learned previously to identify *mybad.host.com* DNS queries as malicious, receive a feature set for a DNS query for *toobad.host.com* and classify it with the same label it would give to DNS queries for *mybad.host.com.* A Machine Learning Agent notifies the network defender via an Observer Agent providing a misuse detection signature ID for reference (see Section 4.5.7 for more information about Observer Agents). At this point the network defender needs more context, and performs a query against a Protocol Analysis Agent for all recent DNS queries and looks for one similar to *mybad.host.com.* The network defender finds a DNS query for *toobad.host.com* that matches the IP and port numbers reported by the Machine Learning Agent and then further queries an Protocol Analysis Agent to determine if any hosts have made requests for web-pages at the IP address that *mybad.host.com* resolved to. At this point, the network defender is in a better position to decide how to react in real-time to the alert. The network defender can deploy a new misuse detection signature for *toobad.host.com* and indicating that it belongs to malicious multi-agent system communications, or whitelist it depending on the functioning mode of the Multi-Agent Malicious Behaviour Detection system, or enable some form of traffic manipulation (Section 4.5.6) to attempt to interact with any malicious software agents associated with *toobad.host.com* in a manner that does not further threaten the protected network.

Protocol Analysis Agents subscribe to Traffic Source Agents to retrieve network traffic, perform some degree of session reconstruction, and maintain some representation of the network environment that the network defender can query against to aid in identifying threats in real-time. For the purposes of this research, two Protocol Analysis Agents were implemented. These are described in Section 5.5.5.

In Figure 4.1, Protocol Analysis Agents subscribe directly to Traffic Source Agents and produces information about the target protocol. The protocol information is then published to Observer Agents and Traffic Manipulation Agents.

4.5.6 Traffic Manipulation Agents

In computer network security, it is often advantageous to modify existing network traffic in order to give network defenders an advantage over the malicious multi-agent systems. One might want to disrupt malicious multi-agent system communications, or interact with malicious software agents to elicit specific behaviours. Traffic Manipulation Agents are responsible for modifying existing network traffic or generating new network traffic. Two primary opportunities for agents to manipulate the network traffic are discussed here. First, Traffic Manipulation Agents can interrupt malicious multi-agent system communications to render the malicious software agents ineffective in the network. This is intended as a quick reaction capability to allow a response (either from a network defender or some other intelligent system) when a malicious software agent is recognized. Second, Traffic Manipulation Agents can influence the network environment to provide more exposure for the Machine Learning Agents to known or recently discovered malicious multi-agent system behaviours.

The first opportunity involves Traffic Manipulation Agents hindering malicious software agents. Suppose, for example, the Machine Learning Agents identify an HTTP request to a known malicious web server. In this case, if the Multi-Agent Malicious Behaviour Detection system is capable of identifying the request fast enough, it can reset the connection from the host and prevent that host from receiving a potentially malicious reply. The technique requires that Traffic Manipulation Agents craft reset packets based on observed traffic. While difficult, is not uncommon among various commercial security products, such as those discussed in Section 2.4. Another technique involves identifying DNS requests for malicious hosts and responding to them locally with a loopback address. If a malicious software agent depends on a successful DNS request and response to locate controlling infrastructure in a malicious multi-agent system, this technique can render malicious software agent unable to communicate with the malicious multi-agent system controllers. For traffic other then HTTP or DNS, Traffic Manipulation Agents can change firewall rules on the fly to reject communications from malicious multi-agent systems. While the Traffic Manipulation Agent does not remove malicious software agents it provides time for some other entity to consider the current status and react to it appropriately. In the implementation described in Chapter 5, the entity responsible is the network defender - however, the framework itself supports the addition of other agents that could take on this responsibility.

The second opportunity involves promoting controlled interactions with malicious multi-agent systems to help train Machine Learning Agents. There is a weakness in the Multi-Agent Malicious Behaviour Detection system in that, even though the Machine Learning Agents are capable of discovering previously unknown malicious multi-agent system communications they still must be based on something that the system has seen before - even if something only remotely similar. The Multi-Agent Malicious Behaviour Detection system requires at least some exposure to malicious multi-agent system traffic in order to learn to detect the target malicious multi-agent system behaviours (Section 1.5.4).

Some exposure can be provided by artificially injecting previously recorded traffic through Traffic Source Agents, discussed in Section 4.5.1. When previously recorded traffic is not available, Traffic Manipulation Agents can be used to elicit infection in a controlled way. A Traffic Manipulation Agent can interact with malicious multi-agent system by mimicking the same behaviours that the Multi-Agent Malicious Behaviour Detection is designed to detect, such as beaconing or updating (Section 1.5.4). The goal of the interaction is to elicit a response from the malicious multi-agent system, which may include recovering a sample of the malicious software agent. Traffic Manipulation Agents are similar in concept to the Honey Monkey in Wang et al. [2006], the Drone Manager in Gu et al. [2007] and Nepenthes in Baecher et al. [2006] described earlier in Section 3.6. The difference is that, while the systems previously mentioned collect malware, Traffic Manipulation Agents are part of a larger system capable of interacting with malicious multi-agent system to enable learning by other agents in the system. A Traffic Manipulation Agent is tasked to perform some potentially risky action tailored to elicit interaction of infection from a specific malicious multi-agent system. Figure 4.2 and Figure 4.3 illustrate an example scenario. In Figure 4.2 a Traffic Manipulation Agent makes a request for the URL *www.badstuff.com*, where the red squares represent the inserted packets. Assuming *badstuff.com* is hosting malicious software agent software, the get request will retrieve an infected document and forward the malicious component onto a specific machine purposed for subsequent infection, as in Figure 4.3, where the green boxes represent the response from the malicious server. There are a number of potential dangers in eliciting infection. This research does not specifically address those dangers, I direct the reader to the various research in malware collection reviewed in Section 3.6.

While the functionality provided by Traffic Manipulation Agents in this case is similar to a honeypot, there is an important distinction. A honeypot is a monitored machine that is vulnerable to attack. Such vulnerabilities may include no firewall, easy to guess passwords, unpatched software, older operating systems, etc. Since there is a significant amount of automated victim discovery, a honeypot has the potential to be compromised by automated as well as supervised attacks. The primary disadvantage

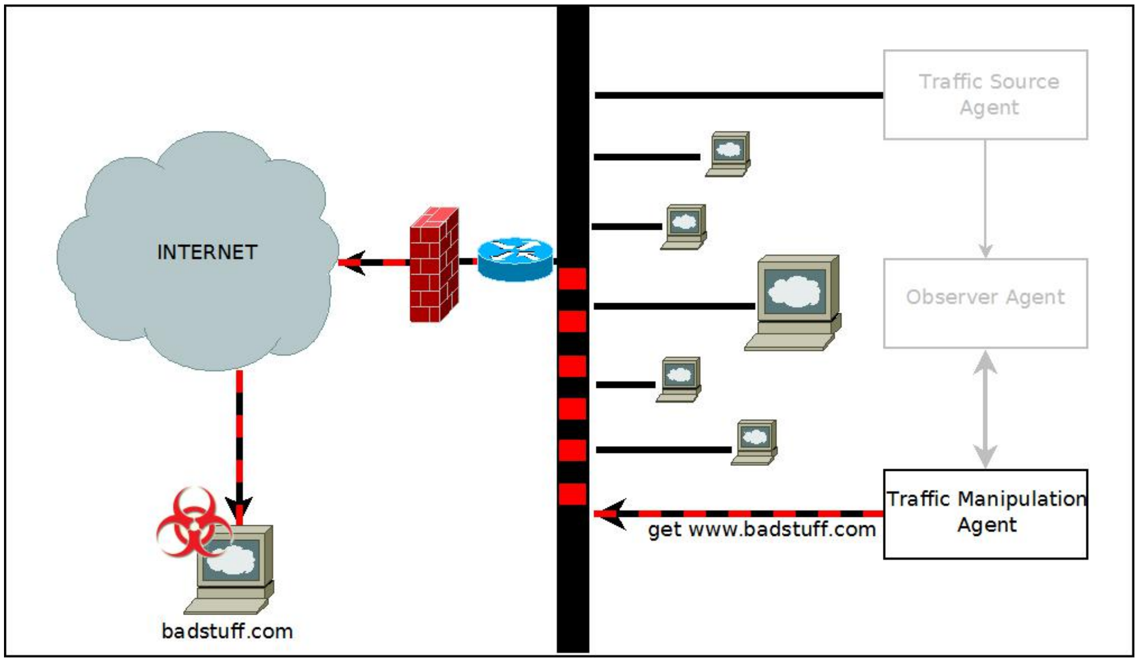


Figure 4.2: A Traffic Manipulation Agent attempts to infect a host by making a web request to badstuff.com.

of a traditional honeypot is that a large number of attacks require user intervention. Many attacks are content delivery, and with no one to access a document infected with malware, the honeypot may never be infected. A Traffic Manipulation Agent, on the other hand, takes a more proactive role in eliciting interaction from malicious multi-agent systems.

Traffic Manipulation Agents can target recently published network attacks given that several sources make it possible to identify websites hosting infected documents. For example, if a specific network attack involves redirecting hosts to a website to download malicious software, and the website is reported in the analysis of the attack, the agent can browse to the website intentionally in order to retrieve the malicious software and infect a controlled host.

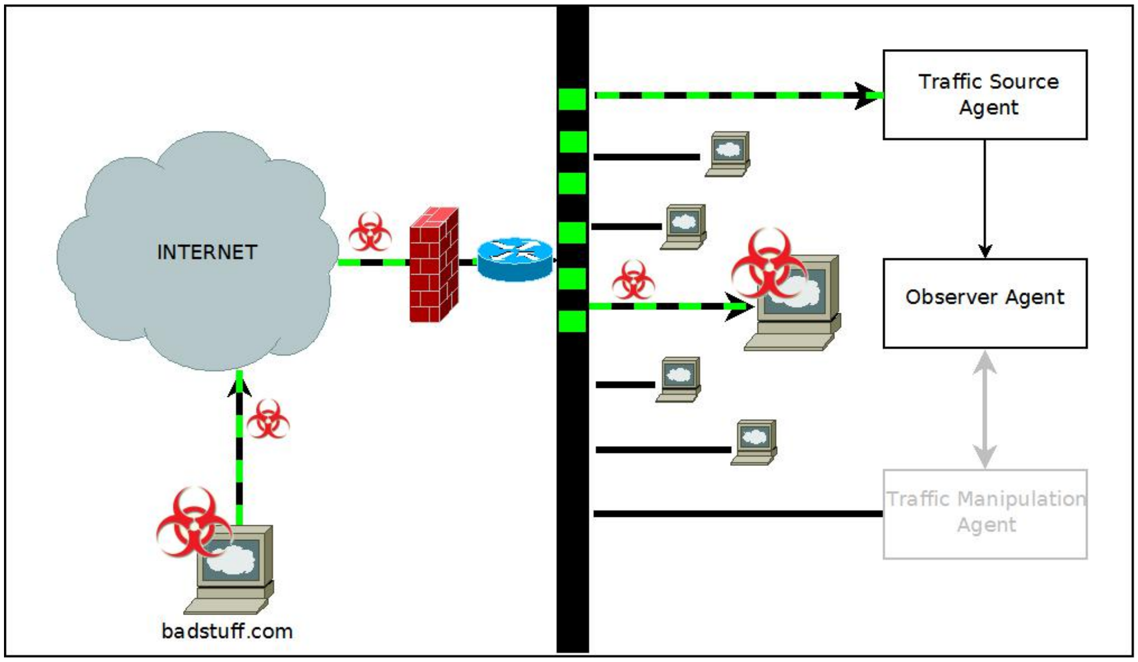


Figure 4.3: Badstuff.com infects the host that the Traffic Manipulation Agent spoofed the request from.

In Figure 4.1 Traffic Manipulation Agents subscribe to Machine Learning Agents, Observer Agents, and Protocol Analysis Agents. Traffic Manipulation Agents use the information derived from those agents as perceptions and plans based on those perceptions. Traffic Manipulation Agents could, for example, attempt to masquerade as one of the protected hosts in an attempt to communicate with malicious software agents in a malicious multi-agent system. This leads to learning through exploration: as the Traffic Manipulation Agents (with human network defender assistance) actively explore interactions with malicious software agents, the Machine Learning Agents can learn from the malicious multi-agent system responses. The communications between Observer Agents and Traffic Manipulation Agents are important. Traffic Manipulation Agents tasked with maintaining communications with a malicious multi-agent system may require intervention to maintain communication integrity between the local malicious software agent and its controlling malicious multi-agent system, ensuring the malicious multi-agent system does not become suspicious of the infected box and reject it. Additionally, the communications must remain both secure and unmodified to prevent the malicious multi-agent system from exploiting them if it does become suspicious. As network defenders learn more about the malicious multi-agent system that a Traffic Manipulation Agent is interacting with, they can add routines to handle novel malicious multi-agent system behaviour, expanding the Multi-Agent Malicious Behaviour Detection system's capabilities. This intervention on behalf of the network defender requires a robust Observer Agent supported by a teleautonomous system. Similar to work performed in [Wegner, 2003], the Traffic Manipulation agents are essentially semi-autonomous.

4.5.7 Observer Agent

Previously, I introduced the Protocol Analysis Agents, responsible for providing human network defenders with abstract information about specific protocols. Each Protocol Analysis Agent is capable of deriving those abstract features from a single protocol, caching them until either a network defender (or some other agent) makes a request for the information, or until the cached information ages off. However, while Protocol Analysis Agents provide specific protocol information, there is still a requirement for defining how human network defenders interact with the framework as a whole. While there is a motivation for a completely autonomous Multi-Agent Malicious Behaviour Detection system, human input is still required to enhance the

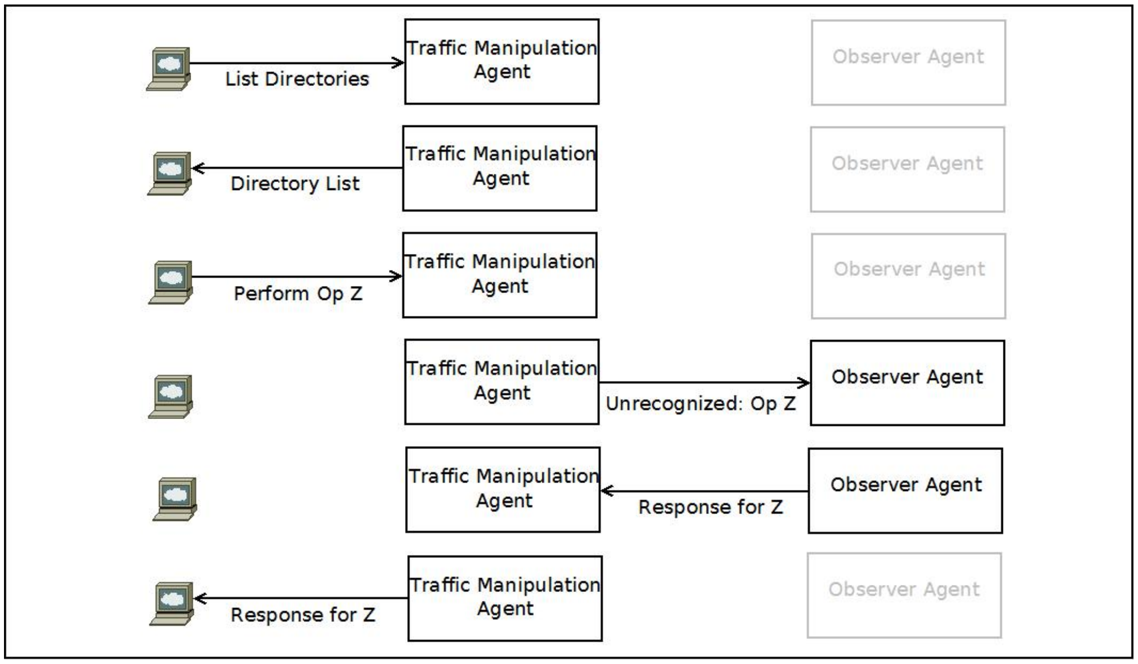


Figure 4.4: A Traffic Manipulation Agent tries to carry on communications with a malware server. When the malware server makes a request that the Traffic Manipulation Agent does not understand it notifies an Observer Agent. The network defender, interacting through the Observer Agent, provides a suitable reply.

capabilities of various agents. Observer Agents provide the interface between the network defender (Section 4.4) and the Multi-Agent Malicious Behaviour Detection system. Observer Agents consume one or more sources of information, whether it be from Traffic Source Agents, Protocol Analysis Agents, Alert Source Agents or Traffic Manipulation Agents, and provide the network defender with a means to configure or interact with those agents. The goal of Observer Agents is to provide situational awareness to network defenders and the capability to interact with the Multi-Agent Malicious Behaviour Detection system. Observer Agents act as the mechanism for providing network defenders a degree of teleautonomous control over the Multi-Agent Malicious Behaviour Detection Agents.

Observer Agents are effectively user interfaces. While user interfaces are not the primary focus of this work, an interface is obviously an important part of any computer system expected to interact with humans. The architecture requires an ergonomic user interface, and there is one important point in the user interface literature that is imperative to follow. The user must always be in control [Krogseter et al., 1994; Krogseter and Thomas, 1994; Strachan et al., 2000]. The Observer Agent places the ultimate control in the hands of the network defender. The network defender is able to make decisions on how the Multi-Agent Malicious Behaviour Detection system operates and can interact with Multi-Agent Malicious Behaviour Detection Agents at any time. Because of this, each Multi-Agent Malicious Behaviour Detection Agent implementation must provide provisions for a network defender to take control or shutdown any Multi-Agent Malicious Behaviour Detection Agent instance. This is especially important for Traffic Manipulation Agents (Section 4.5.6), which may themselves inadvertently perform malicious actions against machines owned by innocent bystanders. There is always the risk that while working with malicious multi-agent systems in an Internet connected environment, a malicious software agent may get out of control and do harm or damage to any connected networks.

In Figure 4.1, Observer Agents accepts messages from multiple Multi-Agent Malicious Behaviour Detection Agents. In order to provide a high degree of situational awareness for the network defender, multiple sources of information are consolidated in Observer Agents.

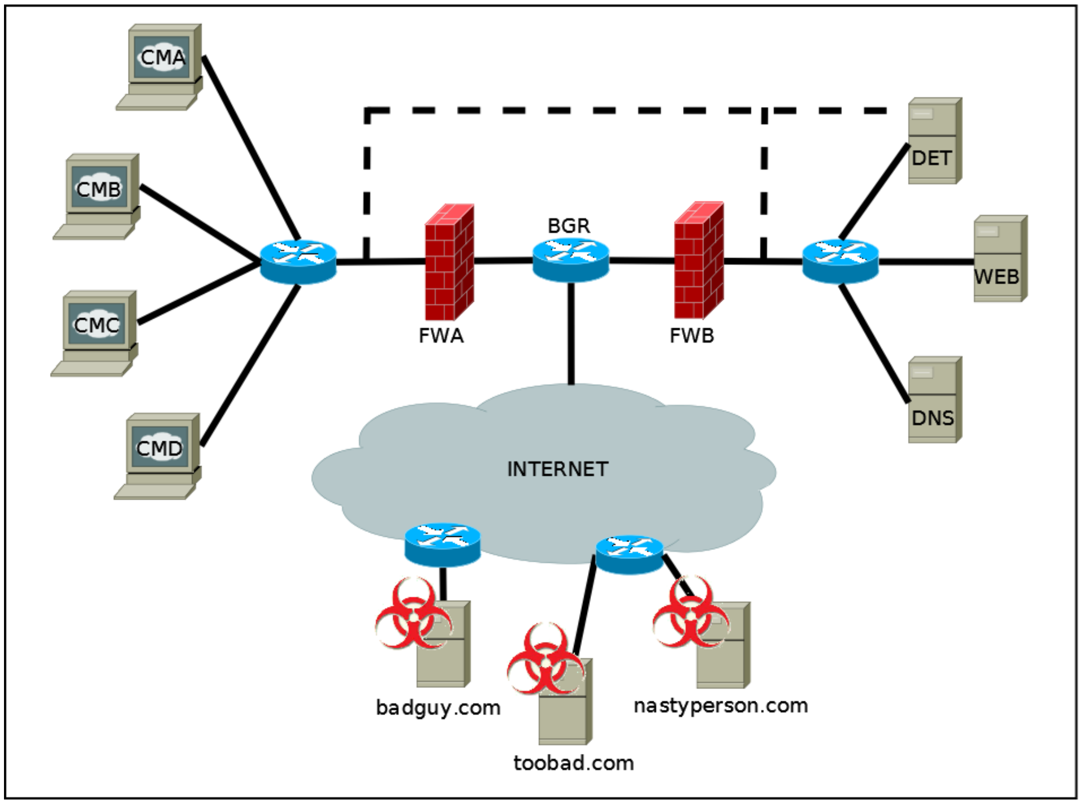


Figure 4.5: A fictional network consisting of a client network, a server network, and an Internet connection.

4.6 Agent Interactions

The previous section described the various Multi-Agent Malicious Behaviour Detection Agent types in this architecture. Each Multi-Agent Malicious Behaviour Detection Agent performs some form of communication with one or more other Multi-Agent Malicious Behaviour Detection Agents (or with human network defenders). In any multi-agent system, however, the important element of the system as a whole is the interaction between agents and the behaviour that arises as a result. Thus, before moving to an implementation-level discussion of these agents (Chapter 5), I focus on an example of a collection of the agents described in this chapter, deployed in a fictional network setting. This setting is depicted in Figures 4.5 to 4.13, and described in the subsections that follow.

4.6.1 Network Environment

The network in Figure 4.5 is divided into three segments connected by a border gateway router, designated *BGR*. Left of BGR is a client network, right of BGR is a network of servers, and below BGR is a connection into the Internet.

The network of client machines contain four hosts: *CMA*, *CMB*, *CMC*, and *CMD*. Between the border gateway router and the network of clients is a firewall, labelled *FWA* (Section 2.4.1).

The network of servers is relatively simple. It consists of a DNS server (*DNS*) and a web server (*WEB*). In addition to the servers there is a firewall (*FWB*) between the border gateway router and the servers. A misuse detection system system (DET) is connected via a passive network tap, represented by the dotted lines, to the link entering both the client network and the server network (Section 2.4.2).

In Figure 4.5, the Internet is represented by a cloud. Three infected machines are shown connected to the cloud via generic border gateway routers. These machines represent a portion of a malicious multi-agent system. The malicious multi-agent system in this scenario is seeking to recruit additional hosts for distributed denial of service attacks (Section 1.5.4).

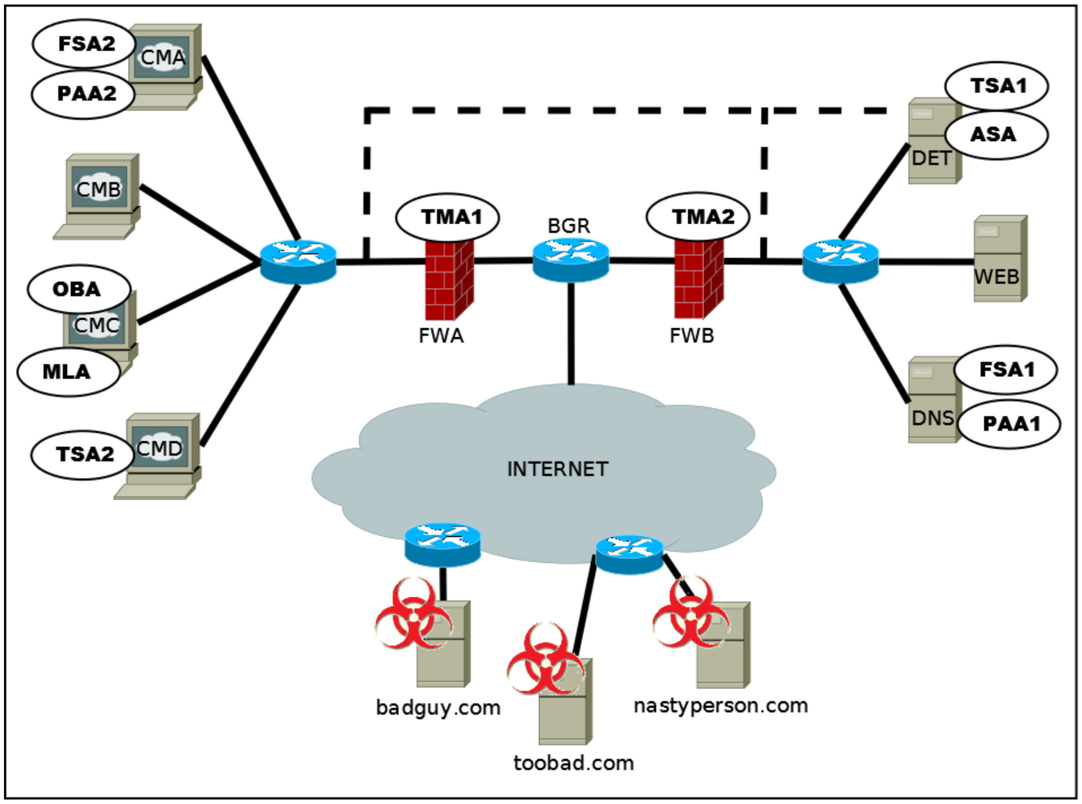


Figure 4.6: An example deployment of agents in a fictional network environment.

4.6.2 Agent Deployment

Figure 4.6 shows a possible Multi-Agent Malicious Behaviour Detection system deployment. A Traffic Source Agent (TSA1) and an Alert Source Agent (ASA) share resources with the DET. This is an ideal position for both agents, as TSA1 can monitor traffic into both subnetworks and the ASA can read alerts from the Snort intrusion detection system. A Feature Source Agent (FSA1) and a Protocol Analysis Agent (PAA1) reside on the DNS server. The client network hosts a number of different agents. CMA hosts both a Feature Source Agent (FSA2) and a Protocol Analysis Agent (PAA2). CMD has a resident Traffic Source Agent (TSA2). CMC is sharing resources between a Machine Learning Agent (MLA) and an Observer Agent

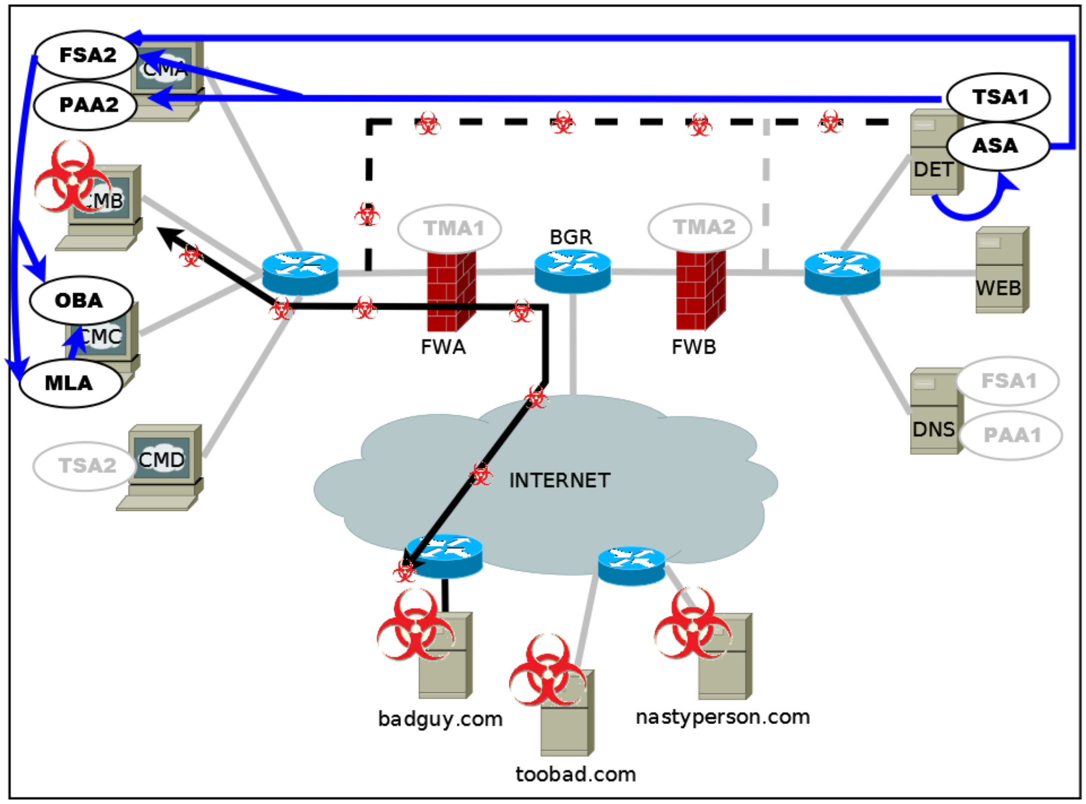


Figure 4.7: Client Machine B is infected by *badguy.com*.

(OBA). A network defender would be physically present at CMC, interacting with the Multi-Agent Malicious Behaviour Detection system via the user interface provided by the OBA. Two Traffic Manipulation Agents (TMA1 and TMA2) are also present in the network, one on each of the firewalls. From this position, the Traffic Manipulation Agents can inject packets into the network and interact with the firewalls to block malicous traffic leaving the network.

4.6.3 Initial Infection

Now that the network setting has been described, I will run through four scenarios. Note that these situations have been greatly simplified compared to actual network scenarios, given that in a real world network scenario hundreds or thousands of feature sets could traverse the network in a matter of minutes. In Figure 4.7, CMB makes a web request out to an infected web server *badguy.com*, and is served an infected pdf document. TSA1 observes each packet between CMB and *badguy.com*, recognizes that FSA2 is interested in packets to any of the client machines on port 80 and sends messages containing those packets to FSA2. As FSA2 receives the packets, it builds up a feature set. Also, PAA2 has registered an interest in web traffic and receives the packet messages from TSA1 as well. In parallel, DET fires an alert on one of the packets between CMB and *badguy.com*. ASA reads the alert, checks if it is relevant to Multi-Agent Malicious Behaviour Detection, and prepares an alert message to notify any interested agents in the system. FSA2 receives the alert message and matches it to an existing feature set it is currently building. When FSA2 receives the FIN packets between CMB and *badguy.com*, it finalizes the feature set and sends it to MLA and OBA. MLA processes the feature set.

At this point the packets between CMB and *badguy.com* have impacted the Multi-Agent Malicious Behaviour Detection system in a number of ways. PAA2 caches the web request and response headers, so that network defenders (or other agents) can access this information until it expires from the PAA2 cache. OBA notifies the network defenders that a session has been identified by DET as containing possible malicious multi-agent system communications. MLA has learned to identify some form of malicious multi-agent system communication based on the feature set it received from FSA2.

At this point, the network defender is aware that some infection may have taken

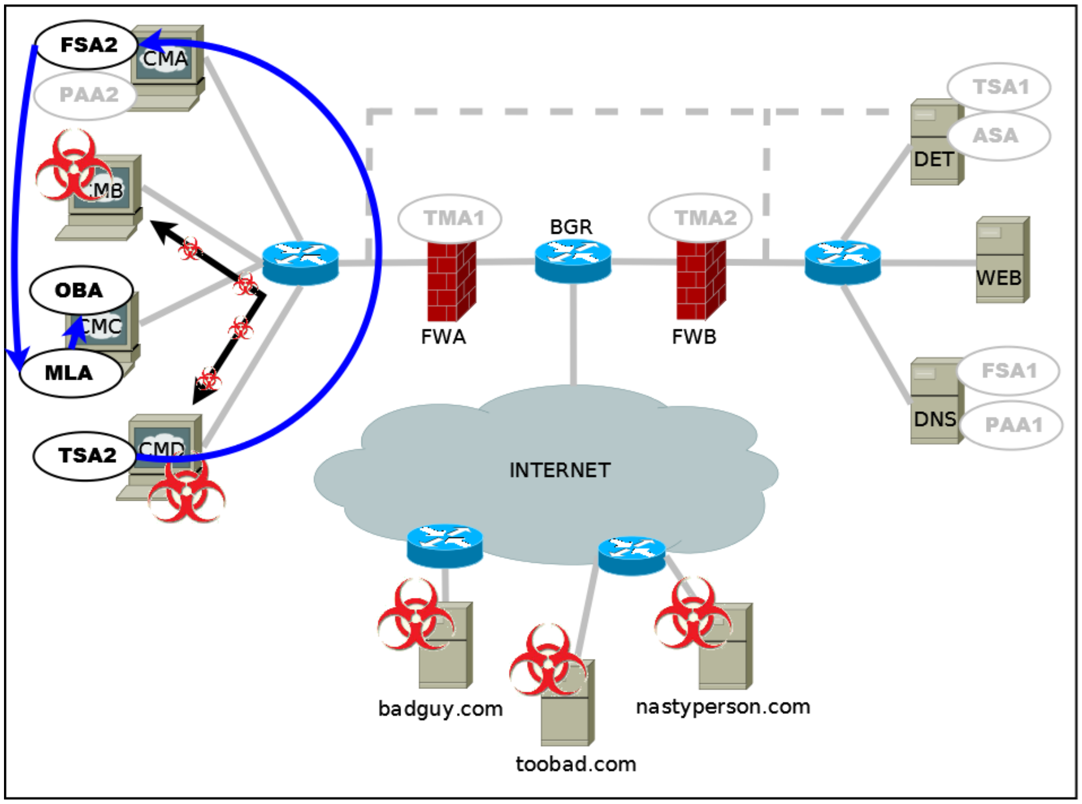


Figure 4.8: The malicious multi-agent system propagates to Client Machine D.

place. However, here I assume that the network defender decides to take advantage of the Multi-Agent Malicious Behaviour Detection system to learn more about the potential infection. In this scenario CMB, now that it is infected, will attempt to partake in three behaviours: propagate to local hosts, beacon out to the malicious multi-agent system infrastructure, and update to the latest malicious software agent version.

4.6.4 Propagate

Suppose the malicious multi-agent system in question is capable of taking advantage of local file shares to propagate to other hosts. The malicious software agent on CMB initiates a connection to CMD, and copies an infected file over to CMD (Figure 4.8). TSA1 does not observe the packets, as they do not leave the local network past the network tap that feeds TSA1. However, TSA2, which is resident to CMD, observes the packets for the session and identifies that FSA2 is interested in all packet between client machines. TSA2 generates a series of packet messages and sends them to FSA2. FSA2 builds a feature set for the observed packets. Unlike the initial infection, no alert message is associated with the feature set, so it is passed to MLA unlabelled. MLA attempts to classify the feature set, and finds that it shares similar features with the previous feature set involved in the initial infection. MLA notifies OBA that is has detected behaviour similar to the initial infection.

Now the network defender has information regarding possible propagation. There is an opportunity for the network defender to feedback into the Multi-Agent Malicious Behaviour Detection system by writing a new misuse detection signature that increases the likelihood of catching the propagation should it occur again.

4.6.5 Beacon

Figure 4.9 shows the two infected clients on machines CMB and CMD. The malicious software agent are likely to try and contact the malicious multi-agent system infrastructure using a form of beacon. In the example illustrated by Figures 4.9 to 4.12, CMB makes a DNS request for *badguy.com*. The request goes to the local DNS server, and returns with a response. CMB, using the IP address from the response, makes an HTTP request to the server *badguy.com* and receives a single packet reply.

TSA1 observes the DNS request and response packets, generating packet message

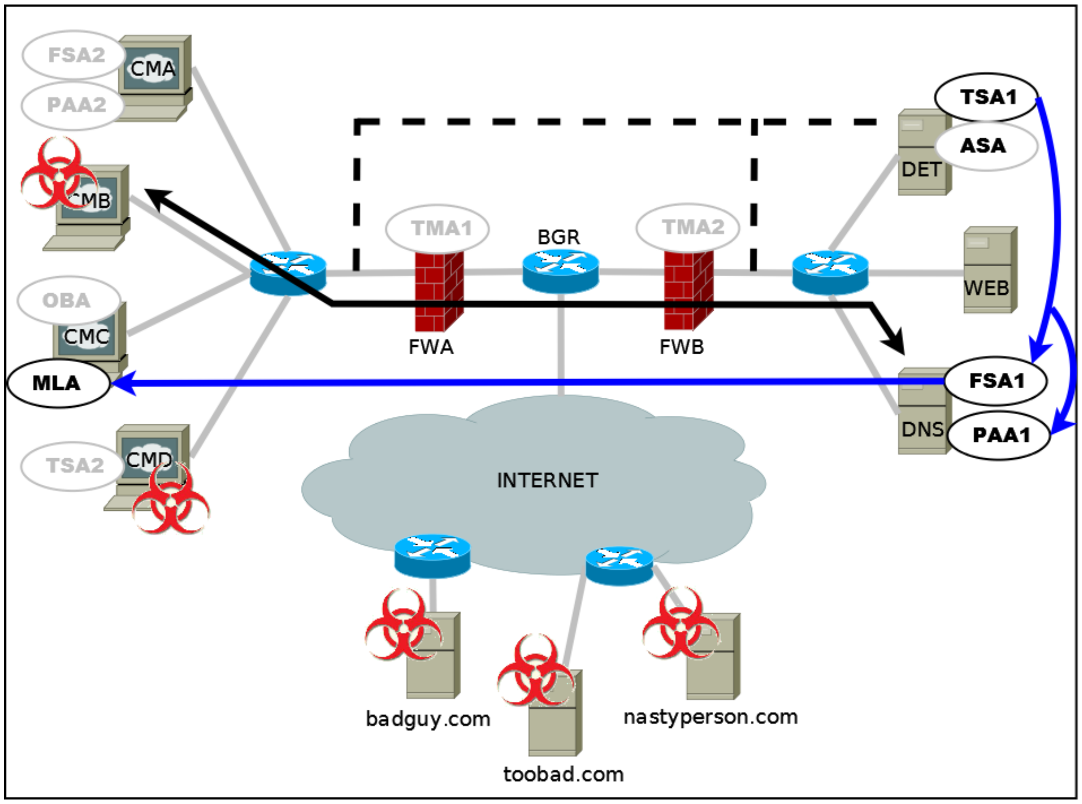


Figure 4.9: Client Machine B performs a DNS lookup for *badguy.com*.

for both. The TSA1 determines that FSA1 is interested in DNS traffic and sends packet messages to FSA1. Equally, PAA1 has also registered interest in DNS packets, and therefore also receives the packet messages. PAA1 caches the request and response, making it available to the network defender via the OBA. FSA1 generates feature sets for the DNS traffic and passes it off to the MLA, that in turn classifies it as a benign DNS lookup (Figure 4.9).

The packets involved in the HTTP request and response are observed by TSA1, and passed to both PAA2 and FSA2 (Figure 4.10). PAA2 caches the HTTP headers, making them available to the network defender. FSA2 generates a feature set for the HTTP session and passes it on to MLA. MLA attempts to classify the feature

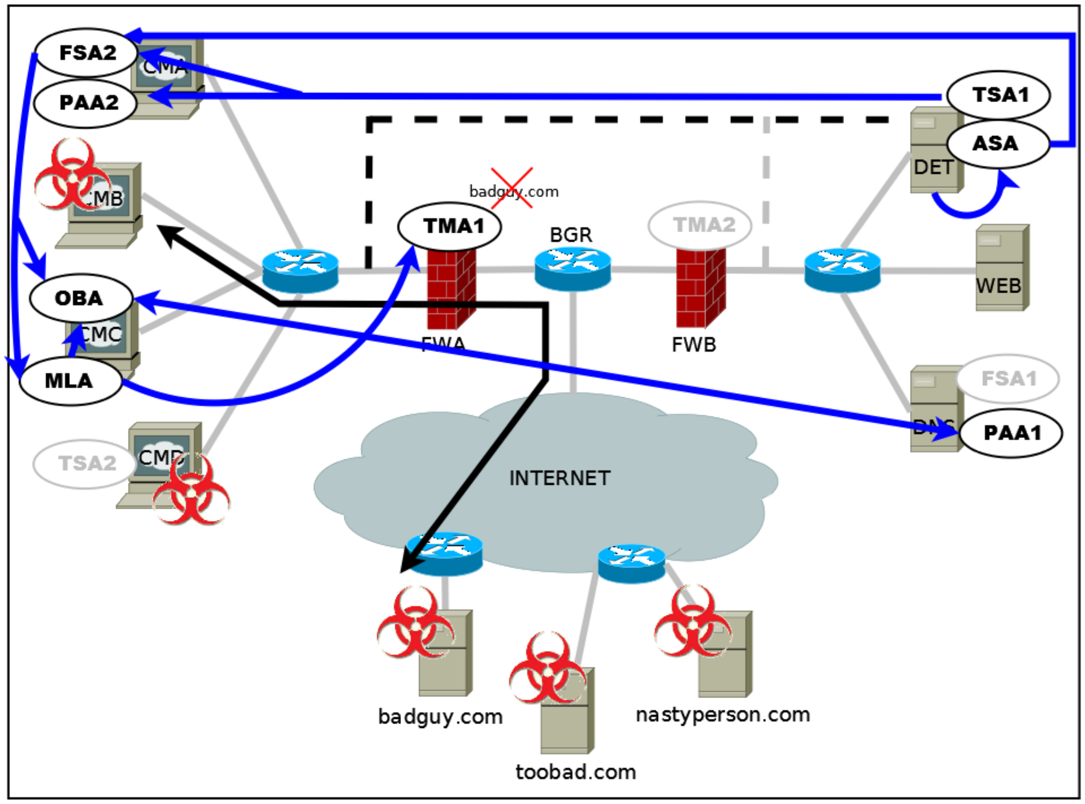


Figure 4.10: Client Machine B beacons to the malicious multi-agent system infrastructure.

set, and finds the features similar to a beaconing technique witnessed by another malicious multi-agent system. Both OBA and TMA1 are notified as they have both registered interest in beaconing behaviour. TMA1 adds the IP address to a blacklist, and interfaces with the FWA, adding a rule to block all attempted connections to the IP address. It also notifies the network defender via the OBA. The network defender receives notification from the OBA of a possible beacon as well as the notification from TMA1 that it has taken action to block further communications to that IP address. The network defender makes a query via the OBA for DNS requests that have resolved to the IP address that TMA1 has now blocked. The network defender is

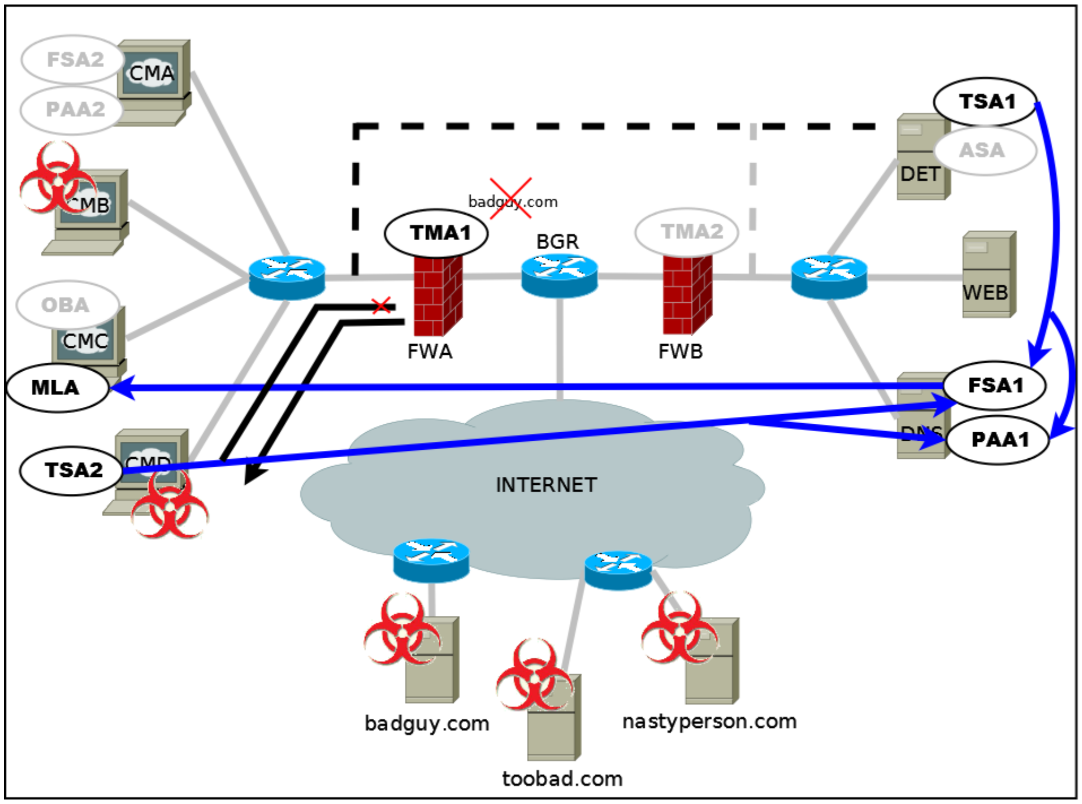


Figure 4.11: Client Machine D attempts a DNS request for *badguy.com*.

informed by PAA1 that there was a recent DNS request for *badguy.com* that resolved to the IP address in question. The network defender can then react by writing a misuse detection rule for DNS requests to *badguy.com* and indicating to TMA1 that if it observes further requests to *badguy.com* it should generate a DNS response indicating that *badguy.com* actually resolves to the same IP address as WEB.

In Figure 4.11 CMD makes a DNS request for *badguy.com* and TMA1 answers the DNS request, redirecting the CMD to WEB (Figure 4.12). Effectively, the Multi-Agent Malicious Behaviour Detection system has disabled the beaconing behaviour.

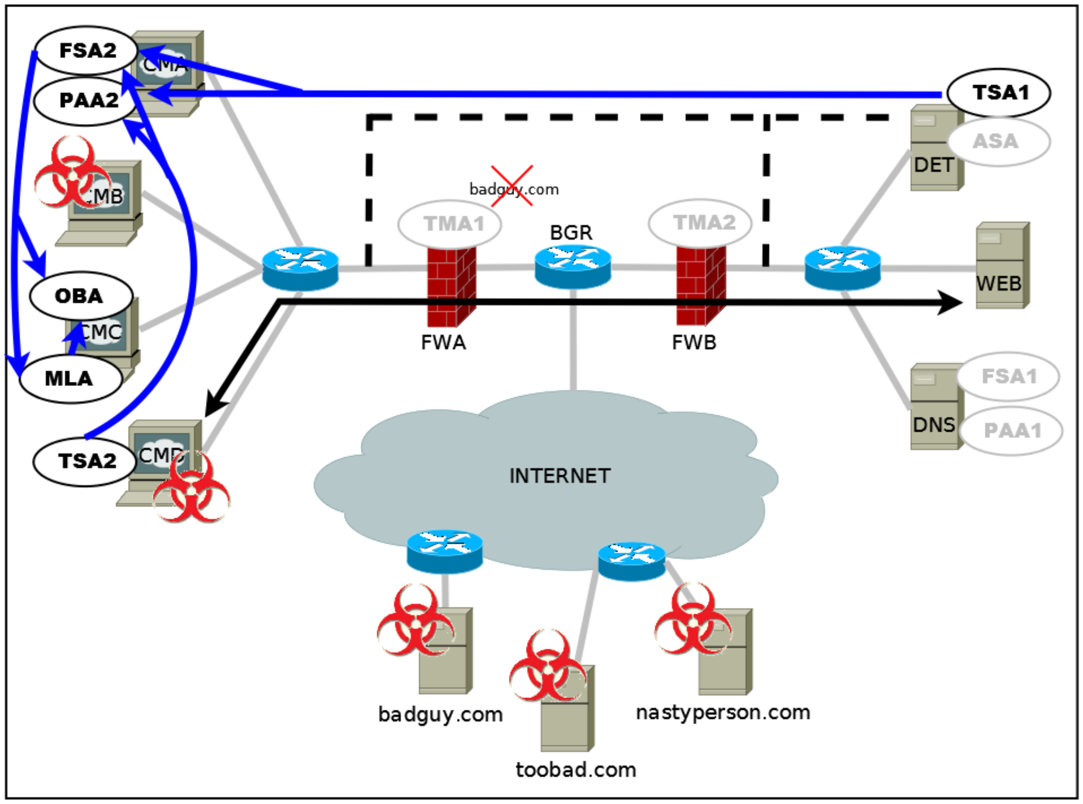


Figure 4.12: Client Machine D is redirected to WEB.

4.6.6 Update

Suppose that the malicious software agent have a routine that indicates they should attempt to update on a fixed interval. However, unlike the beacon described in the last section, the update attempts to retrieve an executable from *nastyperson.com*. Figure 4.13 illustrates the updating behaviour. CMB makes a DNS request and receives a response, the resulting traffic will be processed through the Multi-Agent Malicious Behaviour Detection system as a benign DNS request, similar to how the DNS request for *badguy.com* was treated in the last section (Figure 4.9). When CMB attempts to retrieve an update, via an HTTP request to the IP address hosting *nastyperson.com*, the misuse detection system does not fire an alert, as the misuse

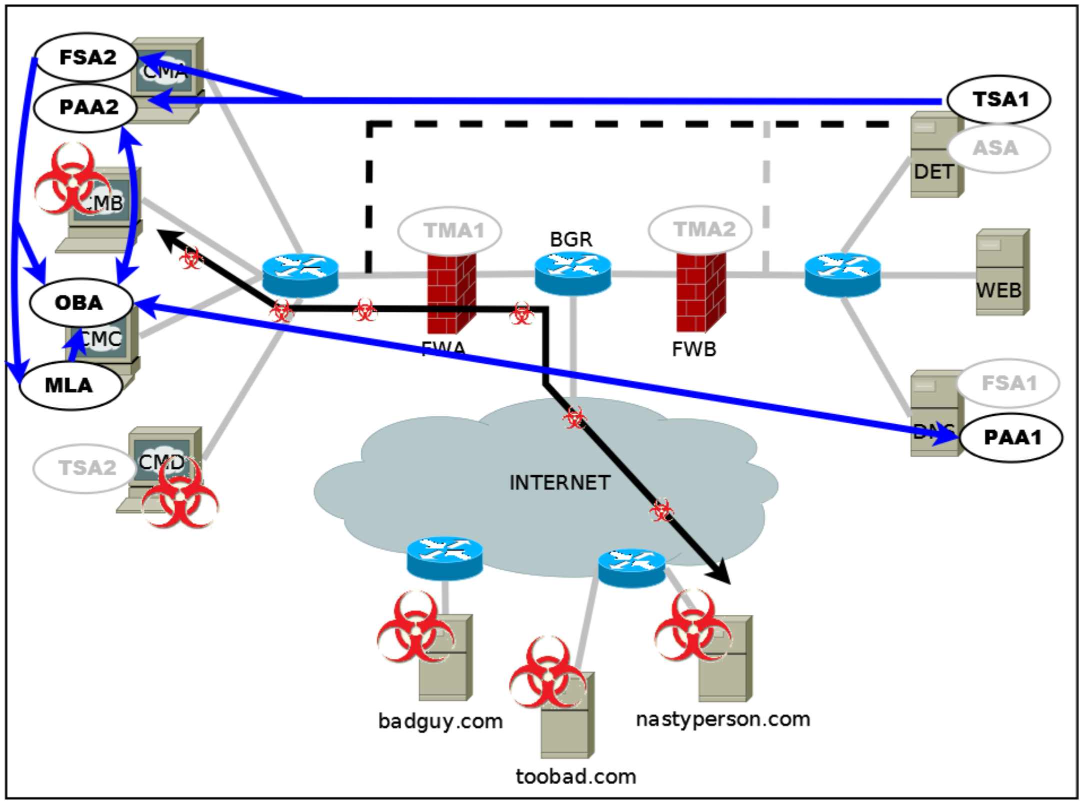


Figure 4.13: Client Machine B attempts to retrieve an update from *nastyperson.com*

detection signature does not succeed in matching against the packet due to changes in the executable returned by *nastyperson.com*. However, TSA1 observes the packets, and sends packet messages to the interested agents, FSA2 and PAA2. FSA2 produces a feature set for the HTTP request and response and passes it off to the MLA. MLA attempts to classify the feature set, and based on previous learning during the initial infection (4.6.3), classifies the feature set as potential malicious multi-agent system communications. The network defender is notified via the OBA, does some investigation by interfacing with PAA1 and PAA2, and uses the unique identifier on the classified feature set to match the potential malicious multi-agent system communications to the misuse detection signature in the misuse detection system knowledge base. Now the network defender can again feedback into the Multi-Agent Malicious Behaviour Detection system by developing a misuse detection signature.

The examples above only explore a subset of the possible interaction between the agents within the Multi-Agent Malicious Behaviour Detection system. However, these simple scenarios show the novel aspects of this system as a whole:

- Multi-Agent Malicious Behaviour Detection agents are deployed across a number of hosts, distributing the resource load across a number of processor cores: improving resilience of the entire system, and taking advantage of idle processor cycles.

- Machine Learning Agents generalize from specific malware detection signatures to broader malicious multi-agent system behaviour, assisting in discovering previously unknown malicious behaviour.

- Protocol Analysis Agents provide the network defender with information required for further investigation, through an Observer User Interface enabling real-time detection.

- Traffic Manipulation Agents redirect malicious software agents to benign services, enabling further study, while still allowing them to perform some malicious behaviours.

- The amalgamation of several Multi-Agent Malicious Behaviour Detection agent functions into a one purposed multi-agent system capable of malicious multi-agent system detection.

While the example is not exhaustive, it should give the reader a fairly complete understanding of the various agent interactions in the framework.

4.7 Summary

This chapter described the Multi-Agent Malicious Behaviour Detection system architecture. The various agents and their functions were discussed as well as the mechanism that supports the agent communications. Finally, I gave an example of how a series of sessions could be processed by a hypothetical Multi-Agent Malicious Behaviour Detection system deployment. The next chapter will provide a detailed description of the implementation of the Multi-Agent Malicious Behaviour Detection system for those researches seeking to implement their own variation or replicate the work here.

Chapter 5

Multi-Agent Malicious Behaviour Detection Implementation

5.1 Overview

To validate the abilities of the framework described in Chapter 4, I implemented a Multi-Agent Malicious Behaviour Detection system, and then evaluated the implementation by deploying it on an experimental network. This chapter describes the implementation of the Multi-Agent Malicious Behaviour Detection framework, both to provide insight in interpreting the environment in which the experiments of Chapter 6 were performed, and to offer some insight for those interested in replicating this work. I begin with a description of the target environment and the programming languages used to implement the various agents. Next, I discuss the communication framework deployed to maintain agent interactions. Finally, I describe the implementation of agents in the system and discuss issues encountered during the implementation of the Multi-Agent Malicious Behaviour Detection system. Further low-level implementation details of all components are available as appendixes, which will be referred to at various points in this chapter.

5.2 Target Environment

There are a number of different network environments into which the Multi-Agent Malicious Behaviour Detection framework could be embedded, such as IPX, AppleTalk, SNA or even UMTS. However, the dominance of TCP/IP makes it an obvious choice for attacks based on malicious multi-agent systems, and an equally obvious choice to implement a system for detection of such attacks. Therefore, I have targeted the implementation of the Multi-Agent Malicious Behaviour Detection framework to local area networks implementing the TCP/IP model of the Internet. Further, the Multi-Agent Malicious Behaviour Detection Agents I implemented only perform detection at the Internet, Transport and Application layers, ignoring the Link layer. While other implementations or future work may consider the Link layer, it was not considered a significant enough source of malicious multi-agent system activity to warrant further attention from this research.

Multi-Agent Malicious Behaviour Detection Agents were implemented on either Microsoft Windows 7 or Mac OS X platforms. This decision was based on the availability of the machines and my relative comfort with the development environments of each, but part of the point of this choice was to illustrate that the framework is not confined to one particular operating system environment. The agents could have been implemented on other platforms, such as Unix, Linux, or FreeBSD.

5.3 Implementation Language

I chose to implement the Multi-Agent Malicious Behaviour Detection Agents using three different computer languages: C#, C++, and Python.

The majority of agents were implemented in C#. The decision to use C# was influenced by its ease of use and the richness of the Microsoft Dot Net Libraries. As a language C# contains many object-oriented features that make development using it attractive. The development of Multi-Agent Malicious Behaviour Detection agents naturally fits with the object-oriented paradigm, and C# is a powerful object-oriented language.

The disadvantage of using C#, as opposed to C or C++, is speed. While this was a more prominent issue five to ten years ago, many limitations of interpreted languages (such as Java and C#) have been overcome by more intelligent just-in-time compilation and improvements in interpretation. There are still methods for optimizing C++ that make it attractive over C# for some tasks, but the overall ease-of-use of a language like C# makes it competitive for rapid object-oriented development.

In order to take advantage of some pre-built machine learning libraries (Section 5.5.3), the Machine Learning Agents described in Section 4.5.3 were implemented in C++. C++ is a powerful language. However, given that it is essentially built on top of a non-object-oriented language (C) and still provides access to the features of C, it is less attractive for use in agent implementation.

Finally, Python provides a mechanism for prototyping agents quickly. Python is an impressive scripting language. It is geared toward object-oriented development and, like C#, is attractive for agent development.

I am confident that Multi-Agent Malicious Behaviour Detection Agents could be implemented using a variety of languages. It was my intention to choose a set of languages that allowed flexibility in implementation decisions and suited the task at hand, while serving to demonstrate the concept of agent design being independent of language choice. The abstraction of the Multi-Agent Malicious Behaviour Detection Agents from the underlying network and communication infrastructure (Section 5.4.1) ensures that developers could potentially implement additional Multi-Agent Malicious Behaviour Detection Agents and integrate them with the existing agents implemented as part of this research in order to pursue future research in this area.

All code for this research was designed and implemented with object-oriented techniques. Since the code is object-oriented, it is very extensible, as well as easy to read and understand. Wherever possible, objects related to one another were organized into object hierarchies.

Now that the target environment and implementation tools have been described, I can build upon this by describing the communications used by Multi-Agent Malicious Behaviour Detection Agents.

5.4 Communications Infrastructure

The core communications infrastructure implemented for Multi-Agent Malicious Behaviour Detection required flexible communications, suitable for a number of heterogeneous agents from multiple platforms. The goal was to enable agents to register their interest in specific feature sets, alert types, or raw packets, to other agents across the system. This allows Multi-Agent Malicious Behaviour Detection Agents to seek out the information needed to facilitate malicious multi-agent system behaviour detection.

A few communications mechanisms were considered, such as .NET Named Pipes, C Style Sockets, various shared memory mechanisms and the advanced message and queuing protocol (AMQP). Early prototypes used .NET Named Pipes, but I found certain aspects of the implementation of Named Pipes limiting. Multi-Agent Malicious Behaviour Detection Agents required too much information about their fellow agents to interact with them, such as each agent's IP address, pipe naming conventions and availability. In later implementations, .NET Named Pipes were replaced with Rabbit MQ, an Erlang implementation of AMQP. Rabbit MQ provides a rich set of desirable features for multi-agent communications. The greatest benefit of this is interoperability of messaging irrespective of the implementation language of the agent. While .NET Named Pipes have a simple interface, their simplicity is limited to specific platforms. While it might be possible to enable UNIX-based agents to communicate using .NET Named Pipes, it increases complexity and the solution may lack stability. AMQP offers a specification designed to be interoperable.

5.4.1 AMQP: Communication Models

While it is beyond the scope of this thesis to provide a complete overview of AMQP, I will discuss some relevant details with an emphasis on how I use AMQP in the Multi-Agent Malicious Behaviour Detection system. For more information regarding AMQP refer to springsource [2012]. RabbitMQ is a specific implementation of AMQP, and their website provides detailed information on the AMQP protocol.

RabbitMQ is a message *broker*, it accepts messages from *producers* and delivers them to *consumers*. However, producers do not send messages directly to consumers. Instead, producers send message to *exchanges*. Exchanges are responsible for accepting messages from producers and routing them to *message queues*, where each message queue is associated with a consumer. Each consumer registers a message queue with an exchange indicating what messages they are interested in using a *binding key*. Each time a producer publishes a message they assign it a *routing key*. Exchanges then route messages depending on: the exchange type, the routing key of the message, and the binding key of the consumers associated with the exchange. The model is often referred to as a publish and subscribe model. Producers publish messages and consumers subscribe to them. The exchanges manage the internals of routing the published messages to the proper subscribers.

Exchanges route messages based on four predefined exchange types, namely: direct, topic, fanout and headers. *Direct* exchanges route received messages directly to any queues bound to the exchange where the message's routing key is an exact match to the queue's binding key. *Fanout* exchanges can have multiple queues bound to them, and will route all messages received by the exchange to all bound queues regardless of the routing key of the message or binding key of the queue. *Topic* exchanges allow the binding keys from a queue to contain wild cards. The exchange will deliver a message to all queues whose binding key matches the routing key while taking into consideration the wild card matches. Valid wild card characters at the time of writing are # and *. # matches against zero or more dot-delimited words. * matches exactly one word. For example, a binding key such as *SQUARE.#* would match any messages sent to a topic exchange whose routing key was *SQUARE.RED.DARK* or *SQUARE.BLUE* but not *CIRCLE.RED.DARK*. Additionally, a binding key such as *SQUARE.*.DARK* would match any messages sent to a topic exchange whose routing key was *SQUARE.RED.DARK* or *SQUARE.BLUE.DARK*, but not *SQUARE.RED.LIGHT* or *SQUARE.DARK* or *CIRCLE.RED.DARK*. This thesis does not make use of the *Header* exchange type and it is therefore not discussed further.

There are a number of exchanges declared in the Multi-Agent Malicious Behaviour Detection system. A set of topic exchanges provide Multi-Agent Malicious Behaviour Detection Agents with the capability to publish information about what they have observed on the network. Multi-Agent Malicious Behaviour Detection Agents subscribe to the exchanges using appropriate binding keys to ensure they get the information that is valuable to their particular role. While exchanges can be dynamically allocated, the Multi-Agent Malicious Behaviour Detection system uses the following eight primary exchanges:

Packet A very basic topic exchange allowing agents to subscribe to packets collected by a Traffic Source Agent. Agents subscribe to packets using binding keys with a source IP address, a source port, a destination IP address, a destination port and a protocol. When subscribing to a Packet exchange, the subscribing agent receives the raw packet data, starting from the IP layer header continuing down to the application layer data. Binding keys should consist of the hexadecimal string representation of each of the five elements making up the key. For example, an agent that requires all TCP data between IP addresses 192.168.1.1 and 192.168.1.2 on port 80 binds a queue to the Packet exchange with binding key

" C4A80101.0050.C4A80102.*.06". The IP addresses are represented as eight characters, 2 characters per byte of the IP address. Port numbers are represented by 4 characters, 2 characters per byte, finally the protocol is represented by 2 characters for a single byte. All exchanges in the Multi-Agent Malicious Behaviour Detection system use a similar scheme for representing numbers as byte strings.

PacketSummary This exchange is for agents that do not require the complete raw packet. Agents publishing to this topic exchange provide a subset of the headers for the packet or some small amount of feature information. For example, the packet length. Subscribing to the Packet Summary exchange requires the exact same binding key format as the Packet exchange binding keys. The exchanges are different in the amount of data they accept from publishing agents and provide to consuming agents. The Packet Summary is used when the packet data is not required by the subscribing agent.

Alert A topic exchange agents use for publishing alerts generated by either behaviour or rule based detection mechanisms indicating malicious or otherwise interesting behaviour. Agents responsible for monitoring traffic for specific alert types can publish using routing keys comprised of an alert ID, source IP address, source port, destination IP address and destination port. For example, an agent can subscribe to all alerts with alert ID 170, source IP address 192.168.1.1, source port 80, destination IP address 192.168.1.2 and destination port 3089 using the following binding key:" 000000AA.C4A80101.0050.C4A80102.0C11".

Feature A topic exchange for publishing traffic features. Traffic features contain information about packets transferred between two IP addresses during a specific session, typically described by a five tuple. The traffic features, as the name implies, contain a set of features. For more information on traffic features refer to 5.5.2. Agents can subscribe to the Feature exchange using a binding key containing the five elements of the standard network five tuple: the client IP address, the client port, the server IP address, the server port and the protocol. There are a couple of important differences between the Feature exchange and the Packet exchange. First, the Packet exchange deals with single packets, whereas the Feature exchange deals with the features of a series of packets between two IP addresses. This highlights the second difference. The Packet exchange uses source and destination, since there is only one packet with a single direction. Traffic features are described in terms of client and server. For simplicity sake, the server is always the side of the communication with the lowest port number. While this is not always the case, typically lower port numbers indicate the server role in a session. An alternative implementation involves setting the initiator of the session as the client, and the server as the host that responds by sending back a packet, regardless of the port numbers.

Labelled The Labelled exchange is a destination for the features set after they've been labelled by the Machine Learning Agents. Both Observer Agents and Traffic Manipulation Agents are likely to subscribe to the Labelled exchange.

AgentControl All agents wishing to participate in the system subscribe to the AgentControl exchange. The AgentControl exchange enables agents to broadcast messages to one another. This exchange is designed to provide agents with a mechanism to explore more complex behaviours. For instance, an agent might signal other agents that it exists and what type of information it is interested in receiving. An agent can also announce if it is shutting down. All agents implemented in this research check the agent control exchange for messages as part of their sense cycle. The exchange is fanout.

AgentLog When Agent logging is enabled, all Multi-Agent Malicious Behaviour Detection Agents periodically publish their current status and some relevant diagnostic information to the AgentLog exchange.

Protocol The protocol exchange is reserved for messages between network defenders and protocol agents, although it could also be used for messages between any agent in the system and a protocol agent. For example, the network defender can request IP address to host name or host name to IP address resolution using the Protocol fanout exchange. Any agents providing some protocol query functionality subscribe to the Protocol exchange and handle messages using a remote procedure call methodology. For an example relating to how DNS is exploited by the Multi-Agent Malicious Behaviour Detection system see 5.5.5.

Filter Traffic Source Agents subscribe to the filter fanout exchange. The filter exchange enables agents to request specific types of traffic from Traffic Source Agents. For example, if the system is intended to study only port 80 traffic, the agent interested in the port 80 traffic can request that the filter be applied to all incoming traffic via the filter exchange. Reducing the amount of uninteresting traffic passed through the system can improve the systems overall efficiency. Filter requests are passed down as five tuples, similar to the topics passed to the Packet exchange. The difference between the Packet and Filter exchange is that, while Packet exchanges allow agents to subscribe to particular types of traffic, Filter exchanges request that a particular type of traffic is not collected at all. For example, an agent might also request that traffic from a particular list of IP addresses is ignored, because the traffic is known to be benign.

5.4.2 AMQP Example Scenario

Having described the communications infrastructure, the following example demonstrates how the agents described in Chapter 4 communicate with one another. The example scenario consists of seven sessions, each containing 31 packets, passed through a hypothetical Multi-Agent Malicious Behaviour Detection system. The example is meant to clearly illustrate how information flows through the system. The sessions contain a series of three DNS requests and responses, three HTTP sessions and a Telnet session. The agents involved in this example include a Traffic Source Agent, an Alert Source Agent, a HTTP Feature Source Agent, a DNS Feature Source Agent, a Transport Layer Feature Source Agent, a Machine Learning Agent, an Observer Agent and a Traffic Manipulation Agent. Figure 5.1 illustrates the agents in the example scenario, as well as the exchanges they are subscribing or publishing to.

The following listing contains a tcpdump-style summary of the packets. All IP addresses are private to ensure that no legitimate IP addresses associated with real world machines were used in the example.

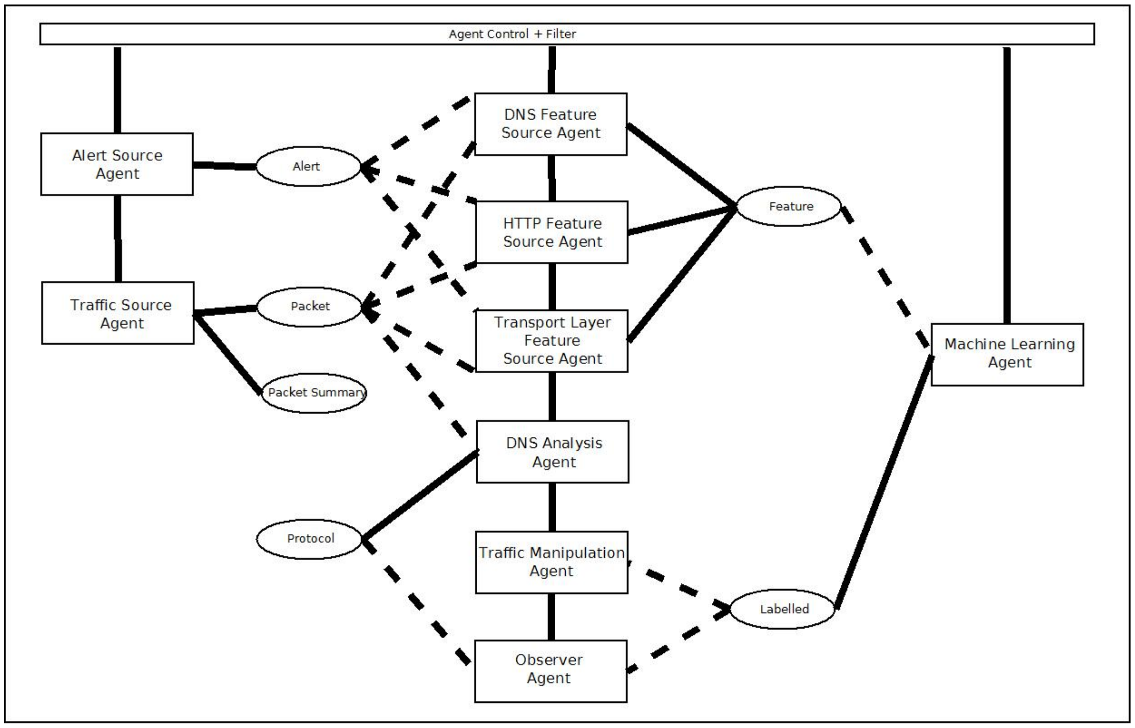


Figure 5.1: Agents in an example scenario. The solid black lines indicate messages published while broken lines indicate a subscription. Ellipses are exchanges and rectangles are agents. The horizontal solid lines connect all agents back to the Agent-Control and Filter exchanges.

1. 192.168.0.1:1045 –>192.168.64.64:53 UDP: badguy.com
2. 192.168.64.64:53 –>192.168.0.1:1045 UDP: badguy.com is 10.1.1.1
3. 192.168.0.1:1046 –>10.1.1.1:80 TCP SYN
4. 10.1.1.1:80 –>192.168.0.1:1046 TCP SYN ACK
5. 192.168.0.1:1046 –>10.1.1.1:80 TCP ACK
6. 192.168.0.1:1046 –>10.1.1.1:80 TCP PSH: GET `badguy.com/1400BytesOfBank Details`
7. 10.1.1.1:80 –>192.168.0.1:1046 TCP PSH RST

8. 192.168.0.1:1047 –>192.168.64.64:53 UDP: benign.search.engine.com
9. 192.168.64.64:53 –>192.168.0.1:1047 UDP: benign.search.engine.com is 10.2.2.2
10. 192.168.0.1:1048 –>10.2.2.2:80 TCP SYN
11. 10.2.2.2:80 –>192.168.0.1:1048 TCP SYN ACK
12. 192.168.0.1:1048 –>10.2.2.2:80 TCP ACK
13. 192.168.0.1:1048 –>10.2.2.2:80 TCP PSH: GET `benign.search.engine.com/search+example`
14. 10.2.2.2:80 –>192.168.0.1:1048 TCP ACK: 1460 bytes of search results
15. 192.168.0.1:1048 –>10.2.2.2:80 TCP ACK
16. 10.2.2.2:80 –>192.168.0.1:1048 TCP ACK: 1460 bytes of search results
17. 192.168.0.1:1048 –>10.2.2.2:80 TCP ACK
18. 10.2.2.2:80 –>192.168.0.1:1048 TCP ACK: 1460 bytes of search results
19. 192.168.0.1:1048 –>10.2.2.2:80 TCP FIN
20. 10.2.2.2:80 –>192.168.0.1:1048 TCP FIN ACK
21. 192.168.0.1:1049 –>10.3.3.3:23 TCP SYN
22. 10.3.3.3:23 –>192.168.0.1:1049 TCP SYN ACK
23. 192.168.0.1:1049 –>10.3.3.3:23 TCP ACK
24. 10.3.3.3:23 –>192.168.0.1:1049 TCP PSH: Generic Telnet Banner

25. 192.168.0.1:1050 –>192.168.64.64:53 UDP: nastyperson.com

26. 192.168.64.64:53 –>192.168.0.1:1050 UDP: nastyperson.com is 10.1.1.1

27. 192.168.0.1:1051 –>10.1.1.1:80 TCP SYN

28. 10.1.1.1:80 –>192.168.0.1:1051 TCP SYN ACK

29. 192.168.0.1:1051 –>10.1.1.1:80 TCP ACK

30. 192.168.0.1:1051 –>10.1.1.1:80 TCP PSH: GET `nastyperson.com/1400BytesOfBankDetails`

31. 10.1.1.1:80 –>192.168.0.1:1051 TCP PSH RST

Packet 1, the first packet the Traffic Source Agent processes, is a DNS request for the host name *badguy.com*. The Traffic Source Agent publishes the packet to the packet exchange, where the packet is queued to both a DNS Protocol Agent queue and a DNS Feature Source Agent queue. The DNS Protocol Agent caches the request so that a network defender can potentially query for it later. The Feature Source Agent generates a feature set for the DNS packet and tries to match the five tuple belonging to the DNS packet to a cache of alerts received from the Alert Source Agent. Depending on the mode of operation, *classify all* or *classify just malicious* traffic, the the DNS Feature Source may label the feature set as DNS. For this example, I will assume that the system is set to only label malicious traffic, and thus the unlabelled feature set is published to the feature exchange. A Machine Learning Agent retrieves the feature set from the the queue, and attempts to label it. Once labelled, the feature set is published to the Labelled exchange and both the Observer Agent and the Traffic Manipulation Agent receive it.

The Traffic Source Agent processes Packet 2, a response to the DNS request for *badguy.com*. The packet follows the exact same route as the first packet. DNS has only one packet per feature set, so the DNS Feature Source Agent has published a total of two feature sets.

The third, fourth and fifth packets are the TCP handshake resulting from the connection attempt initiated when the client looking for *badguy.com* received a DNS response. The packets are published from the Traffic Source Agent to the HTTP Feature Source Agent. The HTTP Feature Source Agent initializes a new feature set keyed on the five tuple {192.168.0.1, 10.1.1.1, 1045, 80, 6}, and the three packets associated with the TCP handshake contribute to the feature set. The feature set will remain in memory until it is timed out, a packet with the reset TCP flag set (RST packet) is received, or the TCP connection is torn down by FIN packets. RST packets are used in TCP sessions to indicate a session should be abandoned and is used for a variety of reasons to terminate TCP sessions instead of a proper connection tear-down using TCP FIN packets.

Packet 6 follows the same path as the TCP handshake into the feature set keyed by {192.168.0.1, 10.1.1.1, 1045, 80, 6}. However, the Alert Source Agent, having also processed the packet, fires an alert indicating that the packet is part of a malicious attack according to the misuse detection system. The HTTP Feature Source Agent receives a message from the alert exchange and matches it to the feature set resident in memory. However, as the session has not timed out, closed or been reset, the feature set remains in memory.

Packet 7 resets the TCP connection associated with {192.168.0.1, 10.1.1.1, 1045, 80, 6}, once the HTTP Feature Source Agent receives packet 7 and matches the packet to the feature set, the HTTP Feature Source Agent publishes the feature set. A Machine Learning Agent receives the feature set and given that there is an alert associated with the feature set the Machine Learning Agent trains with it.

When packet 8 and 9 (the DNS request and response for *benign.search.engine.com*) are captured by the Traffic Source Agent, they follow through the system in exactly the same way as the DNS request and response for *badguy.com*. These produce two feature sets for a Machine Learning Agent as well as a cached entry in the DNS Protocol Agent for *benign.search.engine.com*. To recap, each packet is published from the Traffic Source Agent to the DNS Feature Agent and DNS Protocol Agent. The DNS Feature Agent immediately generates a feature set for each packet and publishes them to the Machine Learning Agent.

Packets 10 through 20 represent a standard HTTP session where a client performs a GET request to *benign.search.engine.com* containing two generic search terms. The server at IP address 10.2.2.2 responds with packets 14, 16 and 18. All of the packets in this session are published through the Feature Source Agent to the HTTP Feature Source Agent, and stored in a feature set keyed with {192.168.0.1, 10.2.2.2, 1048, 80, 6}. Assuming each packet arrived within the timeout period, the feature set remains with the HTTP Feature Source Agent until packet 20, when the second FIN in the session is received.

The Traffic Source Agent processes packets 21 to 24 in a similar manner to packets

10 through 20. However, the HTTP Feature Source Agent is not interested in these particular packets, as they do not meet the port 80 or port 8080 heuristic. Instead, the packets are published to a Transport Layer Feature Source Agent responsible for handling TCP sessions. The packets contribute to a feature set keyed by {192.168.0.1, 10.3.3.3, 1049, 23, 6}. Notice, however, that there is no RST or FIN packet received after packet 24 for this particular session. In this case the feature set will remain in memory until, during a periodic check, the Transport Layer Feature Source Agent recognizes that no packets for the session have been seen for some time and the feature set times out. Once timed out, the feature set is published to the Machine Learning Agents.

Packets 25 and 26 pass through the system again, just as previous DNS request and responses.

The TCP handshake (27 to 29) contributes to a feature set keyed by {192.168.0.1, 10.1.1.1, 1051, 80, 6} in the HTTP Feature Source Agent. Packet 30 and 31 also contribute to the feature set. Once the HTTP Feature Source Agent receives the associated feature set, the feature set is published to the Machine Learning Agent. Unlike the feature set keyed by {192.168.0.1, 10.1.1.1, 1045, 80, 6}, the feature set keyed by {192.168.0.1, 10.1.1.1, 1051, 80, 6} was not matched by an alert provided by the Alert Source Agent. When the Machine Learning Agent receives feature set keyed by {192.168.0.1, 10.1.1.1, 1051, 80, 6} it attempts to classify it and finds that with significant confidence the feature set is similar to the feature set keyed by {192.168.0.1, 10.1.1.1, 1045, 80, 6}. It publishes a labelled feature set and the network defender is notified via the Observer Agent. The network defender is asked by the system to examine the session. The network defender uses the misuse detection signature referenced by the label in the feature set, and sees the misuse detection signature is supposed to match on GET requests to the host name *badguy.com*, and that the signature identifies traffic ex-filtrating banking data. The network defender can verify the DNS cache for the entries that have recently resolved to 10.1.1.1, and finds the host names *nastyperson.com* and *badguy.com*. Additionally, the network defender could examine the contents of the GET requests and look for similarities.

Notice that there is a substantial difference between how the Machine Learning Agent identifies something as potentially malicious and how the network defender notices something malicious. A misuse detection signature was written by a network analyst who noticed that GET requests for *badguy.com* were malicious. This is a relatively easy feature for a human network analyst to observe and then communicate to a machine via a human readable misuse detection signature. The Machine Learning Agent does not look for the same features. Instead the Multi-Agent Malicious Behaviour Detection system translates the features of a session into a format that is suitable for a machine learning algorithm to process. In this example, those features are elements like the average packet length, number of packets in the session, presence of a reset, and entropy of a URL - all features that human security analysts are less likely to recognize. The Machine Learning Agent then matches what it thinks is malicious and grounds the results of the match to a misuse detection signature that approximates its hypothesis that the network defender can understand.

5.5 Implemented Agents

This section describes Multi-Agent Malicious Behaviour Detection Agent implementations based on the seven agent roles discussed in Section 4.5. Figure 5.2 illustrates the variety of agents implemented to validate this work. I will discuss two Traffic Source Agents: the *Pcap Live Traffic Source* and the *Pcap File Traffic Source.* I will describe the Feature Source Agents, specifically the *Transport Layer Feature Source*, the *HTTP Feature Source*, the *DNS Feature Source* and all of the supporting feature related code. I will provide details related to the implementation of Machine Learning Agents. I will present an example Alert Source Agent, the *Unified Alert Source Agent.* I will describe two Protocol Analysis agents, the *DNS Agent* and the *HTTP Agent.* I will introduce implementation of four Traffic Manipulation Agents, the *DNS Manipulation Agent*, the *HTTP Manipulation Agent*, the *Contact Agent*, and the *Dynamic Firewall Agent.* I will provide details of three Observer Agent Implementations, the *Experimenter Agent*, the *Logger Agent* and the *Network 3D User Interface.*

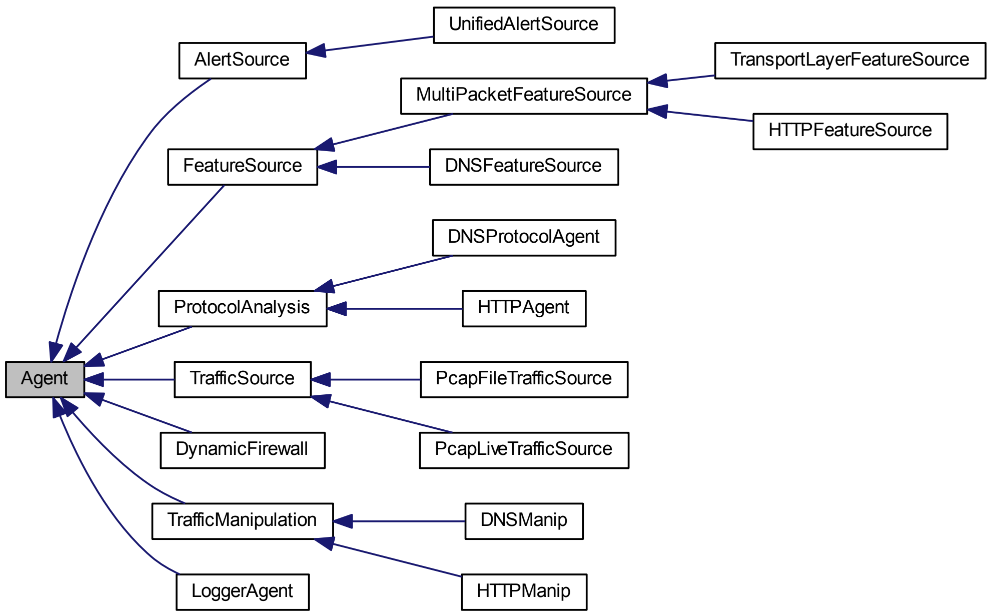


Figure 5.2: Inheritance diagram for the various Multi-Agent Malicious Behaviour Detection Agents.

Significant effort was made throughout the implementation of the Multi-Agent Malicious Behaviour Detection system to ensure that tasks were divided out to autonomous agents capable of existing independent of other agents in the system. This multi-agent concept is strictly adhered to by enforcing an inheritance hierarchy, with each component of the system descending from the Agent abstract class (Appendix A.2). Figure 5.2 shows a variety of agents implemented throughout this work. Throughout this section I will discuss implementation challenges with many of the agents in Figure 5.2.

5.5.1 Traffic Source Agent Implementations

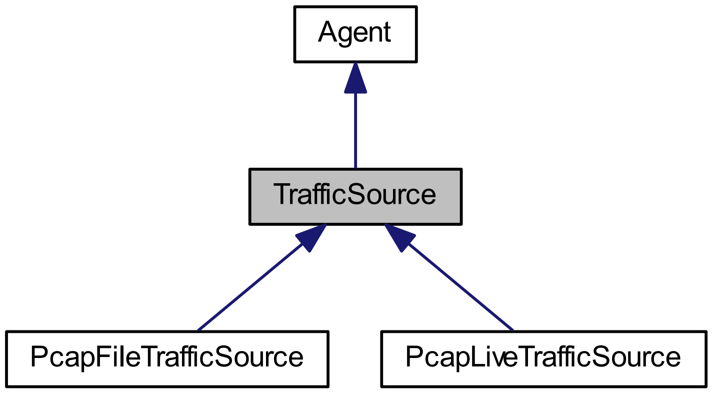


Figure 5.3: Inheritance diagram for the Traffic Source Agents.

As described in Section 4.5.1, Traffic Source Agents are tasked with ingesting raw traffic and publishing it out to Multi-Agent Malicious Behaviour Detection Agents. Traffic Source Agent implementations take advantage of the open source project *SharpPcap* to read packets from either the network interfaces on the host machine or from previously recorded network traffic. SharpPcap is available from Gal [2012]. Effectively, SharpPcap provides an interface into the winpcap drivers for the Microsoft .NET programming framework.

Pcap Live Traffic Source (Appendix A.3) is an implementation of a Traffic Source Agent that reads packets in real time from network interfaces on host machines. In order to mitigate the network speeds, all packets are read into a First-in, First-out queue as quickly as the SharpPcap framework drivers allow. Packets are pulled out of the queue and a limited amount of processing is performed to enable publishing the packets to various destinations for Feature Source Agents (Section 4.5.2) to consume. The queue is monitored to ensure it does not take up too much memory: if the queue becomes too large, new packets are no longer inserted and are discarded instead. Discarded packets are reported to a log. The queue will grow and shrink as the saturation of the network medium changes, and when less traffic is on the network the processing has time to catch up. This introduces the potential for delays in delivery of packets to Feature Source Agents. Given the nature of the work, such delays are difficult to avoid.

Pcap File Traffic Source (Appendix A.3) is an implementation of a Traffic Source Agent that reads previously recorded packet captures. Packets can be read from a previously recorded traffic file at a great speed, which could overload the system with packets. The Pcap File Traffic Source attempts to keep this manageable through a simple timing mechanism is used to introduce an artificial delay between packet reads. Beyond its use in avoiding flooding the system, this artificial delay can also be used to test the Multi-Agent Malicious Behaviour Detection system under various hypothetical network saturation conditions.

The Pcap File Traffic Source can also be used to introduce artificial malicious multi-agent system traffic, perhaps previously recorded on another network, into the Multi-Agent Malicious Behaviour Detection system by running in parallel with a Pcap Live Traffic Source. Feature Source Agents receive the packets through the AMQP server on the Packet exchange (Section 5.4.1) and will not differentiate between traffic sources. A mix of Traffic Source Agents can increase the Multi-Agent Malicious

Behaviour Detection system's exposure to various traffic types while still protecting the host network.

0	2	4	6
dir & type	protocol	label	
client IP		server IP	
client port	server port	packet length	

Table 5.1: Serialized packet summary message. The first row marks byte offsets. The alternating shading marks field boundaries.

Both Traffic Source Agent implementations publish packets in two formats, as *packet summaries* and *full packets*. The format of a packet summary is illustrated in Table 5.1. The packet summary is intended to have a small footprint for performance considerations, consisting of only 20 bytes. The packet summary contains enough information to match it with existing sessions in the Multi-Agent Malicious Behaviour Detection system (client IP and port, server IP and port, and protocol). It also includes a field for a label, should the packet have been matched to an alert message (Section 5.5.4), and the length of the packet. Packet summaries are useful to Multi-Agent Malicious Behaviour Detection Agents that are not concerned with the packet payload. For example, Observer Agents may display flows of data on a user interface, but do not require the packet payload to do so. Full packet messages, illustrated in Table 5.2, are a copy of the packet data from the IP layer down through the application layer. A time stamp, extracted from the winpcap driver, is added to the front of the full packet message.

0	2	4	6
arrival time seconds			
arrival time microseconds			
packet byte data (variable length)			

Table 5.2: Serialized full packet message. The first row marks byte offsets. The alternating shading marks field boundaries.

5.5.2 Feature Source Agent Implementations

My implementation of the Multi-Agent Malicious Behaviour Detection framework required a variety of Feature Source Agents to process some of the features available in network traffic. The inheritance relationships among these are illustrated in Figure 5.4. Recall, Section 4.5.2 identified the challenge associated with identifying the number of packets sufficient for deriving relevant features to send to other Multi-Agent Malicious Behaviour Detection Agents. In this research, the problem of estimating a useful number of packets is handled by a heuristic that divides sessions into three categories: UDP, TCP and DNS. The advantage to the three chosen categories is that they can be swiftly identified with very little processing requirements. Sessions that are identified as TCP are considered complete when a proper TCP connection tear-down is observed, a RST packet is seen, or there is a significant delay from the last packet seen. The delay is heuristically chosen as a multiple of the average round trip time of packets in the session. All traffic on port 53 is presumed to be DNS, and only one packet per session is used for deriving features. UDP sessions rely on a timeout only, but the timeout is aggressive compared to the TCP session timeout, fixed at 30 milliseconds.

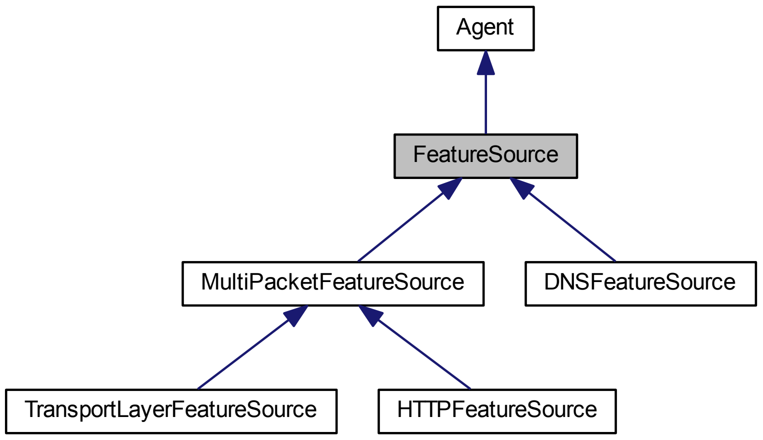


Figure 5.4: Inheritance diagram for the Feature Source Agents.

In addition to the dividing of traffic into UDP, TCP and DNS for determining an appropriate strategy for ending sessions, Feature Source Agents were implemented to extract additional features for DNS and HTTP traffic. With respect to session length, HTTP is treated the same as TCP. To facilitate feature extraction for different session types, features are split across four classes of feature objects (Figure 5.5), whose implementations are described in Appendix A.4. Each *Traffic Features* class represents a single feature set, as described in Section 4.5.2. The Traffic Features class contains a limited number of features available in all session types processed by Multi-Agent Malicious Behaviour Detection, the server IP address, the client IP address, the server port, the client port, the protocol, and the time since the last connection between the client and server IP addresses was observed.

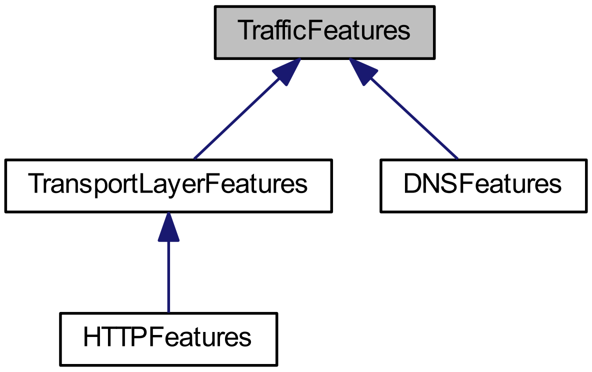


Figure 5.5: Inheritance diagram for feature items.

Given that every feature set object inherits from the Traffic Features base class, each feature set is uniquely identified by a key, consisting of 14 bytes from the the protocol, client IP, server IP, client port and server port. For the purposes of this work, the client IP is assumed to be the IP address associated with the higher of the two port numbers in a given connection. When a Traffic Feature is passed between Multi-Agent Malicious Behaviour Detection Agents, it is serialized into a message whose structure is illustrated in Figure 5.3. The first row in the figure indicates the byte positions for each field.

The Transport Layer Features class extends the Traffic Features class and adds the additional features for sessions of packets, as described in Table 3.1. The format of the Transport Layer Features is illustrated in 5.4. Transport Layer Features are passed between Multi-Agent Malicious Behaviour Detection Agents to describe UDP

0	2	4	6
feature type	protocol	label	
client IP		server IP	
client port	server port	ticks since last connection	
		start time seconds	
		start time milliseconds	
		end time seconds	
		end time milliseconds	

Table 5.3: Serialized Traffic Feature message. The first row marks byte offsets. The alternating shading marks field boundaries.

0	2	4	6
feature type	protocol	label	
client IP		server IP	
client port	server port	ticks since last connection	
		start time seconds	
		start time milliseconds	
		end time seconds	
		end time milliseconds	
		mx md data cnt	q3 data cnt
max data	max data AB	max data BA	req sck BA
max seg AB	max seg BA	min seg AB	min seg BA
max data IP	mx data IP AB	mx data IP BA	mx data IP AB
mean segment size		mean data control AB	
variance data control BA			
total packets		total packets AB	

Table 5.4: Serialized Transport Layer Feature message. The first row marks byte offsets. The alternating shading marks field boundaries.

and TCP sessions, with the exception of DNS.

The feature sets described so far are numerical values that can be extracted and tracked with relative ease. Some of the more complex values, such as the mean segment size or the variance of the data control bytes are slightly more complex for Feature Source Agents to keep track of. However, there are a variety of mechanisms for tracking such values using techniques [Ling, 1983; Chan et al., 1983].

Features of DNS traffic are derived by the DNS Feature Source Agent and stored as DNS Features objects. Since DNS features are derived from single packets, the DNS Feature Source Agent does not require the logic to track multiple packets, and so inherits directly from the Feature Source Agent. Equally, the DNS Features objects do not require the additional features in the Transport Layer Feature class and so inherits directly from the Traffic Features class with the addition of those features described in Table 4.1. One of the DNS features extracted from the DNS packets, are the *host name features.* The host name features are extracted as two distinct features. Each DNS packet should contain a host name query. That host name is a string of characters that the domain name service is asked to resolve to an IP address. Also, the record data portion of a DNS response contains strings the can be exploited for features. In order to derive numerical values from the host name I've introduced a *Network String Features object.* The Network String Features object will consume any string of bytes and return a serialized message of features. Table 5.5 illustrates the features extracted by a Network String Features object, and Table 5.6 illustrates the format of serialized Network String Features. To simplify data passed through the system, each class of features is designed to produce feature sets with a consistent number of features. However, DNS responses and requests can contain a variable number of host names and record data. Instead of providing individual features for each host name or record data, the DNS Feature Source Agent combines all of the host names into one string and all of the record data strings into a second string and extracts two sets of Network String Features to include in DNS Feature sets. Table 5.7 illustrates a DNS Feature message, as it would appear when passed between Multi-Agent Malicious Behaviour Detection Agents.

Network String Features	
Distinct Bytes	The number of distinct byte values in a collection of strings.
Minimum Byte Value	The smallest byte value in a collection of strings.
Maximum Byte Value	The largest byte value in a collection of strings.
ASCII Capital Count	The number of ASCII capital letters in a collection of strings (byte values between 65 and 90).
ASCII Lower Case Count	The number of ASCII lower case letters in a collection of strings (byte values between 97 and 122).
ASCII Digit Count	The number of ASCII digits in a collection of strings (byte values 48 to 57).
Length	The total length of all of the strings combined.
Segment Count	The number of strings in the collection of strings.

Table 5.5: Features extracted from string in HTTP and DNS packets.

0	2	4	6
distinct bytes	byte max & min	upper count	lower count
ascii digit count	total length	segments	
string entropy			

Table 5.6: Serialized Network String Features message. The first row marks byte offsets. The alternating shading marks field boundaries.

The HTTP Feature Agent also takes advantage of the Network String Features object to extract the additional features described in Section 4.2. In order to extract the various fields, the HTTP Feature Source Agent first looks for identifiers that indicate that there is likely an HTTP header in the session, such as GET, POST, HEAD, PUT, DELETE or HTTP. For the sake of efficiency, the Feature Source Agent only examines the first few bytes of the first packet in each session to make a determination. If one of the identifiers are found, the Feature Source Agent will try to parse specific header request or response fields. Fields include the User-Agent, Via, Referrer, Host, Cookie, Server, Location, and Set-Cookie. Additionally, the agent attempts to locate

0	2	4	6
feature type	protocol	label	
client IP		server IP	
client port	server port	ticks since last connection	
		start time seconds	
		start time milliseconds	
		end time seconds	
		end time milliseconds	
		identifier	flags
question count	answer count	name server #	additional rec. #
DNS name (serialized NetworkStringFeatures)			
record data (serialized NetworkStringFeatures)			

Table 5.7: Serialized DNS Feature message. The first row marks byte offsets. The alternating shading marks field boundaries.

the URL. As with DNS, the string features are combined into Network String feature sets. There are three unique string sets for each broader HTTP session feature set: the URL features, the request field features and the response field features. Table 5.8 shows the structure of the serialized HTTP features message. Notice how the message is considerably larger than the other feature set messages described to this point. Serializing the messages helps to reduce the amount of congestion the message cause for the AMQP communications infrastructure (Section 5.4.1).

As described in section 4.5, Feature Source Agents provide a layer of abstraction between raw packets and the features that other agents in the system are interested in. As such, the agents are often part of more complex agents. All feature sources are derived from the FeatureSource abstract class. For more implementation details of the individual features classes see Appendix A.4.

0	2	4	6
feature type	protocol	label	
client IP		server IP	
client port	server port	ticks since last connection	
		start time seconds	
		start time milliseconds	
		end time seconds	
		end time milliseconds	
		mx md data cnt	q3 data cnt
max data	max data AB	max data BA	req sck BA
max seg AB	max seg BA	min seg AB	min seg BA
max data IP	mx data IP AB	mx data IP BA	mx data IP AB
mean segment size		mean data control AB	
variance data control BA			
total packets		total packets AB	
hdr counts	url features (serialized NetworkStringFeatures)		
user agent hdr field (serialized NewtorkStringFeatures)			
via hdr field (serialized NetworkStringFeatures)			
referer hdr field (serialized NetworkStringFeatures)			
	host hdr field (serialized NetworkStringFeatures)		
cookie hdr field (serialized NetworkStringFeatures)			
server hdr field (serialized NetworkStringFeatures)			
location hdr field (serialized NetworkStringFeatures)			
	set cookie hdr field (serialized NetworkStringFeatures)		

Table 5.8: Serialized HTTP Feature Message. The first row marks byte offsets. The alternating shading marks field boundaries.

5.5.3 Machine Learning Agent

As discussed in Section 4.5.3, I chose to leverage a series of existing machine learning algorithms implemented by Saffari et al. [2010]. Those algorithms are Online Random Tree, Online Random Forest, Online LaRank, Online Multi-Class Linear Programming Boost, and Multi-Class Gradient Boost. There are a number of existing machine learning packages that I could have potentially used for this research, including Waffles [Gashler, 2011] or WEKA [Hall et al., 2009]. The reason for choosing Saffari et al. [2010] over another machine learning package was the focus on streaming classifiers provided by the implementations in Saffari et al. [2010].

Machine Learning Agents were implemented in C++ and interface directly with the source code provided by Saffari et al. [2010]. The Machine Learning Agents subscribe to the Features exchange and consume feature sets from the various Feature Source Agents, including the Transport Layer Feature Source, HTTP Feature Source and the DNS Feature Source. The various feature sets are illustrated in Tables 5.4, 5.7, and 5.8.

One Machine Learning Agent is instantiated for each feature set type, since different feature sets contain different numbers and types of features. Given the machine learning algorithms in Saffari et al. [2010] operate on vectors of data represented by float values, each feature set is transformed from a mixed type representation to a vector of floats suitable for the machine learning algorithms. If the received feature set has not been previously labelled, the Machine Learning Agent attempts to classify the feature set. If the received feature set has been previously labelled, the feature set is used to train the machine learning algorithm. Additionally, after training on the labelled feature set, the machine learning algorithm attempts to classify the feature set. The feature set is reclassified to give the Machine Learning Agent a chance to associate a series of confidence values to each feature set it sees.

It is important to Multi-Agent Malicious Behaviour Detection that each feature set is classified, even if it was previously assigned a label by an external misuse detection system, such as Snort. Central to Multi-Agent Malicious Behaviour Detection is the idea that malicious multi-agent systems attempt to mimic benign traffic. The confidence values assigned during classification are exploited to identify malicious multi-agent system communications that are similar to benign traffic, where the highest confidence class is benign but the second or third highest confidence class is malicious.

The confidence values for each class are normalized, such that the sum of all confidence values is 1.0. After classifying a feature set, the Machine Learning Agents publish the feature set with the top five confidence values and associated labels for network defenders and other Multi-Agent Malicious Behaviour Detection Agents to exploit.

Classes are based on the Snort Signature IDs, where each Snort signature ID can be tied back to a type of malicious multi-agent system behaviour, such as beaconing, denying, propagating, ex-filtrating, updating, and mimicking (Section 1.5.4).

5.5.4 Alert Source Agent Implementations

Section 4.5.4 discusses the design of Alert Source Agents, and describes the reliance on external misuse detection to provide context for network defenders. In this research, Snort was chosen as the external misuse detection engine. Snort provides a flexible grammar for describing network activity, applicable to detecting malicious multi-agent system traffic. In particular, the Sourcefire VRT certified rules provide a reference value that can be used to get further information about what specific network activity an alert is designed to detect. In this section I will discuss *Unified Alert Source Agents*. Unified Alert Source Agents integrate Snort into the Multi-Agent Malicious Behaviour Detection system and publish messages providing the signature ID, source IP, destination IP, source port, destination port and protocol for Multi-Agent Malicious Behaviour Detection Agents to ingest. The published messages are then matched to sets of features: any feature set derived from a session that produced a hit in Snort is assigned a label representing the Snort signature ID.

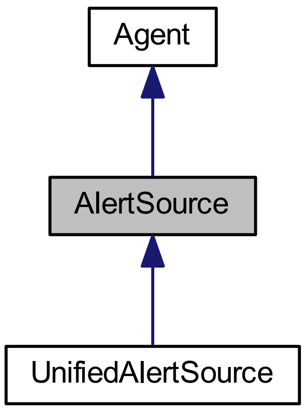


Figure 5.6: Inheritance diagram for Alert Source Agents.

Table 5.6 illustrates the Alert Source Agent inheritance hierarchy. A Unified Alert

Example Key: 000000AA.C4A80101.0050.*.*				
Alert ID	Source IP	Source Port	Destination IP	Destination Port
000000AA	C4A80101	0050	*	*

Table 5.9: Alert Source Routing Key: all messages where alert ID is 170, the source IP is 192.168.1.1 and the source port is 80.

Agent is a C# implementation of an Alert Source Agent that reads Snort *unified alert files* and publishes the alerts to the Multi-Agent Malicious Behaviour Detection system. Snort unified alert files are binary, as opposed to ASCII, formatted files containing 64 byte blocks of data for each alert a Snort process generates in response to some network traffic. The 64 byte block contains a variety of data including the Snort signature ID, a time stamp as well as the source and destination IP addresses, the source and destination ports and the protocol for the traffic associated with the alert. Unified Alert Source Agents exploit the existing Snort misuse detection capabilities, providing the Multi-Agent Malicious Behaviour Detection system an interface into Snort. Appendix A.6 contains the documentation for the Alert Source and Unified Alert Agent implementations. The C# classes were implemented with extendability in mind. For the purposes of the evaluation described in Chapter 6, the Unified Alert Source is the only implemented C# Alert Source. However, its implementation is based on an abstract base class, AlertSource, that can be extended in future work to provide the Multi-Agent Malicious Behaviour Detection system with additional sources of external misuse or anomaly detection.

Unified Alert Agents publish alerts for other Multi-Agent Malicious Behaviour Detection Agents to subscribe to (see 4.5). Unified Alert Agents use the Topic messaging model (Section 5.4.1), where the routing key is devised of five words separated by dots (see Table 5.9 for an example). First word is an alert ID, followed by a source IP address, a destination port, a destination IP address and a destination port. Note that all fields are represented by hex strings, for example the IP address 192.168.1.1 is represented by the string C4A80101. Depending on their requirements, agents can subscribe to any combination of the five features.

In the context of an operating Multi-Agent Malicious Behaviour Detection system, an external Snort process is loaded with a rule set that identifies some malicious multi-agent system behaviours, such as beaconing or propagating (Section 1.5.4). The Snort process generates alerts and writes them to a unified alert file. A Unified Alert Source Agent monitors the unified alert file for changes, and parses new alerts as the Snort process writes them to the unified alert file. The Unified Alert Source Agent then publishes the alerts into the Multi-Agent Malicious Behaviour Detection system. Multi-Agent Malicious Behaviour Detection Agents then match the alert messages from the Unified Alert Source Agent to feature sets derived by Feature Source Agents.

Snort operates at the packet level: it provides some functionality for operating on sessions, but the session mechanism is not currently enabled for this implementation of the Multi-Agent Malicious Behaviour Detection system. A single session will generally consist of multiple packets, and each of those packets could potentially generate a Snort alert. Additionally, a single packet could potentially match multiple Snort rules. With respect to multiple alerts on a single packet, Snort will only generate one alert per packet. Therefore, care must be taken in determining what rules to deploy to ensure that the most relevant Snort rule fires on a packet if multiple rules can potentially fire. Also, if several rules fire across a session, only the most relevant one should be added to the matching feature set that will ultimately be published for Machine Learning Agents. I use a general heuristic for determining which alerts out of a set will be matched to a feature set. Snort rules are ordered such that when two rules match a single packet, the most relevant rule (where relevance is measured by how accurately the rule identifies malicious communications) fires with the highest precedence. When multiple alerts fire on a single session, malicious rules are given precedence over benign rules. For example, if a rule designed to identify all HTTP fires on the first packet in a session, but then a subsequent packet in the same session triggers a malicious detection rule, the malicious detection rule gets precedence and the HTTP rule is ignored.

Network defenders take advantage of Snort signature ID's by comparing output from the Machine Learning Agents, and the associated traffic to validate if the Machine Learning Agent has identified novel malicious multi-agent system communications. The more autonomy the Multi-Agent Malicious Behaviour Detection system achieves, the less reliance there will be on the human network defender to validate its findings. However, at this point human validation is still an important step to ensure proper and safe functioning. Given that the raw output of the Machine Learning Agents is not very meaningful to the network defender, the Snort rules provide a mechanism for the network defender to better understand why the Machine Learning Agent made a particular classification. Also, the network defender can work out why the Snort rule did not fire on the original traffic, but was classified as traffic similar to traffic that typically fires on a specific Snort rule.

5.5.5 Protocol Analysis Agent Implementations

Section 4.5.5 outlined the requirement for a set of Protocol Analysis Agents capable of providing network defenders with contextual information to support their analysis. For this research I implemented two such agents, the *DNS Protocol Agent* and the *HTTP Agent*. Figure 5.7 illustrates the inheritance relationship between the Protocol Analysis Agents described here and the Agent base class.

Monitoring DNS traffic Mockapetris [1987a,b] is important in determining network behaviour and is often exploited by malicious multi-agent system (Section 4.5.5). The DNS Protocol Agent is responsible for interpreting DNS traffic and maintaining a history of DNS activity. Whenever network defenders wish to perform analysis on traffic, they can query the DNS Protocol Agent in real-time to retreive information about recent DNS queries and responses.

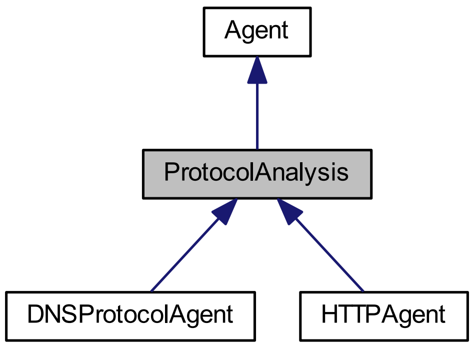


Figure 5.7: Inheritance diagram for Protocol Analysis Agents.

DNS Protocol Agents subscribe to all port 53 traffic from Traffic Source Agents.

Recall from 5.5.1 that agents can subscribe to topics using keys composed of source IP, source port, destination IP, destination port and protocol. DNS Protcol Agents subscribe to UDP traffic using the following AMQP topics: **.0035.*.*.11* and **.*.*.0035.11*. The packets are published through the Packet exchange (Section 5.4.1) to the DNS Protocol Agent's packet queue. DNS Agent pull the port 53 UDP traffic out of the incoming packet queue and begins the process of pulling out relevant information for further analysis. The DNS protocol employs a simple compression mechanism, whereby portions of the domain name are not repeated in a single message. Instead of repeating a domain name, RFC1035 defines a method to replace repeated sections with pointers to the first spot where the domain name appears in the message. While the mechanism reduces network congestion, the pointers are not helpful for analysis. The DNS Protocol Agent removes all compression and stores the DNS requests and responses into custom structures to simplify requests from network defenders for information regarding DNS activity. Network defenders can request a list of host names that have resolved to a specified address, a list of IP addresses that have been associated with a specified host name or a history of requests that a specified IP address has made.

The DNS Protocol Agent maintains a cache of the last ten resolutions per host name and IP address pair. The network defender can take advantage of that information to discover instance of repeated DNS requests, such as might be the case in malicious multi-agent system beaconing activity, or a flood of bogus DNS requests as in malicious multi-agent system denial attack.

HTTP Protocol Agents attempt to maintain some information about various web-pages that clients on the protected network have visited. If the Multi-Agent Malicious Behaviour Detection indicates that there is suspicious HTTP activity (by classifying some HTTP feature set as related to a malicious multi-agent system) the network defender can look up webservers associated with the HTTP activity Fielding et al. [1999]. This allows the network defender to see the names of webpages, documents retrieved, or URLs. The HTTP Protocol Agent parses the HTTP headers to extract the relevant fields and then keeps a history of the field values per webserver observed. At the moment, the interaction is relatively primitive. The network defender submits a web server name and the HTTP Protocol Agent returns a structured list of header field values it has observed. However, future work is planned to extend the capabilities of the HTTP Protocol Agents.

Implementing Protocol Agents is complex, as it requires that the developer understand enough about the protocol in question to parse out important information. However, since the goal of a Protocol Analysis Agent is to extract information relevant to the network defender as opposed to providing complete protocol functionality, heuristics can help to reduce the amount of work and knowledge required by the developer. For example, an SMTP Protocol Analysis Agent could attempt to extract just the email addresses it sees in the traffic, as opposed to attempting to parse the entire email.

Future work is also planned to develop Protocol Analysis Agents that are designed specifically to parse malicious multi-agent system communication protocols. For example, a Protocol Analysis Agent that can recognize and parse Conficker (Section 1.2.2) communications could provide a network defender with an indication of what the malicious multi-agent system is attempting to achieve on the network.

5.5.6 Traffic Manipulation Agent Implementations

Section 4.5.6 described a number of scenarios for Traffic Manipulation Agents. Many of these involve crafting artificial packets. SharpPcap (Section 5.5.1), in addition to capturing packets, enables Traffic Manipulation Agents to place packets back into the network. The *DNS Manipulation Agents*, and *HTTP Manipulation Agents*, implemented as part of this thesis are both responsible for accepting requests from either network defenders or other Multi-Agent Malicious Behaviour Detection Agents to insert packets. The *Dynamic Firewall Agents* interface with network firewalls to block connections. The inheritance diagram for Traffic Manipulation Agents is shown in Figure 5.8. While I group Dynamic Firewall Agents with Traffic Manipulation Agents, since they do not actually insert packets, they inherit directly from the Agent abstract class.

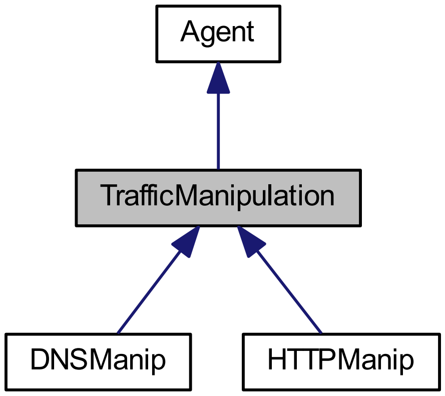


Figure 5.8: Inheritance diagram for Protocol Analysis Agents.

Communications between the Multi-Agent Malicious Behaviour Detection Agents and Traffic Manipulation Agents follow the Remote Procedure Call model (RPC). This model, in the context of AMQP, is described by springsource [2012]. Multi-Agent Malicious Behaviour Detection Agents are responsible for creating anonymous callback queues. When an agent would like to request action from a Traffic Manipulation Agent (such as adding or removing a filter, responding to a DNS request, or closing an HTTP connection), a message is sent with a property set to the name of the callback queue as well as a unique identifier for the request. The Traffic Manipulation Agent will process the requests as they are read from the request queue and send responses to the queue named in the request from the Multi-Agent Malicious Behaviour Detection Agent. The reply contains the request identifier linking it back to the original request. While multiple Multi-Agent Malicious Behaviour Detection

Agents may publish their requests to a single queue read by a Traffic Manipulation Agent, the agent will respond to each Agent's unique queue. The response consists of a status identifying if the action was carried out.

DNS Manipulation Agents subscribe to a packet insertion queue, where messages are sent by other Multi-Agent Malicious Behaviour Detection Agents and network defenders. The messages contain a DNS request packet, as well as a desired response IP address. The DNS Manipulation Agent then crafts a UDP DNS response packet that complies with RFC 1035 and attempts to insert it back into the network. If the Multi-Agent Malicious Behaviour Detection system is fast enough, the DNS packet will reach the requesting machine before the local DNS server can respond. The client then uses the DNS Manipulation Agent's response instead of the actual response. The key to the success of this lies in the specification of RFC 1035, which states that the client making a DNS request should only accept the first legitimate response and ignore all other responses. In effect, the DNS Manipulation Agent is performing a DNS spoofing attack [Yan et al., 2006; Steinhoff et al., 2006]. However, the attack is *on behalf* of the network defender to interact with a malicious multi-agent system. Faking DNS packets is relatively easy, since DNS operates over UDP and UDP is a stateless connection. All one has to know to insert a UDP packet is the IP addresses and ports involved, assuming the application layer is equally simple to mimic.

HTTP Manipulation is more difficult than DNS manipulation, given that TCP connections maintain some state using Sequence and Acknowledgement numbers. However, HTTP is often abused by malicious multi-agent system. Manipulating the data the victim sends out can help protect the client or elicit responses from the malicious multi-agent system, while minimizing the risk to the protected network. The HTTP Manipulation Agent accepts requests to terminate connections. An agent or a network defender can send a message with the last observed packet to the HTTP Manipulation Agent and it will craft a RST packet by determining the next appropriate Sequence and Acknowledgement numbers and sending RST packets to the client and server involved in the offending TCP connection. Additionally, HTTP Manipulation Agents accept URLs and craft the packets necessary to make a web request on behalf of a client machine in the network. Given a URL the HTTP Manipulation will perform a DNS lookup, initiate a TCP connection to the server, create an HTTP GET request, and capture the response.

Dynamic firewall agents are slightly different from the previous two types of Traffic Manipulation Agents. They do no actually insert packets. However, they do modify traffic flow in the network by blocking IP addresses and ports in response to threats to the network. Multi-Agent Malicious Behaviour Detection Agents can send three predefined message types to Dynamic Firewall Agents (see Table 5.10). The messages have varying message and field lengths depending on the desired action. Each message begins with a single byte describing the message type. While only three message types are defined here, the single byte provides for 255 possible messages. The Dynamic Firewall will ignore all message with a value other then 1, 2 or 3. The first type of message, designated by a 1, is the add filter message. The message type is followed by 7 fields; a 4 byte request ID, a 2 byte Multi-Agent Malicious Behaviour Detection Agent ID, a 1 byte protocol, a 4 byte source IP address, a 2 byte source port, a 4 byte destination IP address, and a 2 byte destination port. The fields are all treated

Add Filter Message							
Type	Req ID	Agent ID	Proto	Src IP	Src Port	Dest IP	Dest Port
1 byte	4 bytes	4 bytes	1 byte	4 bytes	2 bytes	4 bytes	2 bytes

Remove Filter Message			
Type	Req ID	Agent ID	Filter ID
1 byte	4 bytes	4 bytes	4 bytes

Check Filter Message			
Type	Req ID	Agent ID	Filter ID
1 byte	4 bytes	4 bytes	4 bytes

Table 5.10: Multi-Agent Malicious Behaviour Detection Agent To Dynamic Firewall Agent messages.

as unsigned integers. The second message type, designated by message type 2, is the remove filter message. Removing a filter requires 4 fields, the first three are similar to the add message, a 1 byte message type, a 4 byte message ID, and a 4 byte Contact Agent ID. The final field is the ID of the filter to be removed. Finally, the Multi-Agent Malicious Behaviour Detection Agent can send a message to verify that the requested filter still exists. The Multi-Agent Malicious Behaviour Detection Agent sends a message identical to the remove filter. However, with a message type set to 3, and waits for a response as to whether the filter with the specified ID still exists.

Dynamic Firewall Agents send reply messages to the Multi-Agent Malicious Behaviour Detection Agent requests (see Table 5.11). In each case, Dynamic Agents send responses to the exchange bound to the queue identified by the Multi-Agent Malicious Behaviour Detection Agent ID sent in the third field of the agent's message. The full response message consists of 3 fields. The first is a message type, which will always be set to 1, indicating a response to a previous request. The second

Dynamic Firewall Response		
Type	Req ID	Response Code
1 byte	4 bytes	2 bytes

Table 5.11: Dynamic Firewall Agent response message.

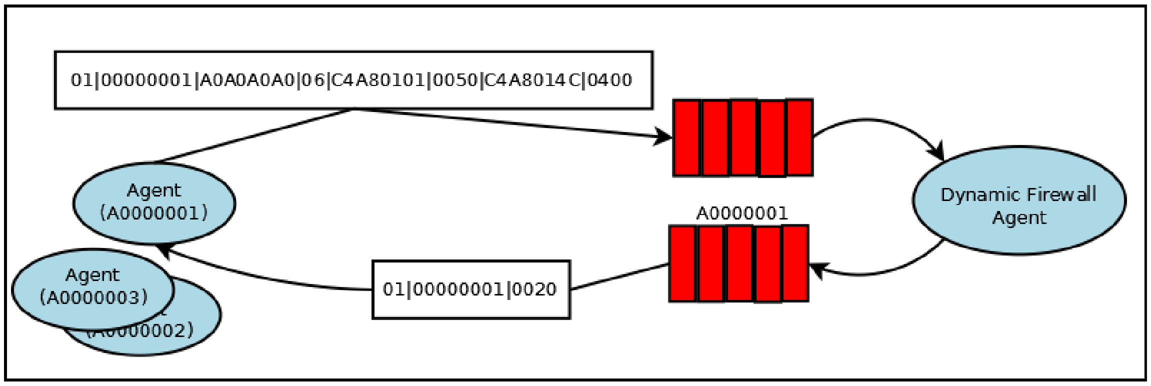


Figure 5.9: Example of a Multi-Agent Malicious Behaviour Detection Agent with ID A0000001 requesting to add a filter blocking all traffic from 192.168.1.1 port 80 to 192.168.1.76 port 1024.

field is a 4 byte request id, so that the Multi-Agent Malicious Behaviour Detection Agent can reliably match each response with a previous request. Finally, 2 bytes indicate the response code of the message. A response code of 0 indicates failure, while a response code greater than 0 identifies the rule ID that was either added or removed. Figure 5.9 illustrates a request by a Dynamic Firewall agent to add a filter to block traffic. The Dynamic Firewall Agent here is implemented specifically for the standard firewall deployed on Mac OSX Lion. I am confident that the existing code can be modified to interface with a number of different firewall and operating system combinations.

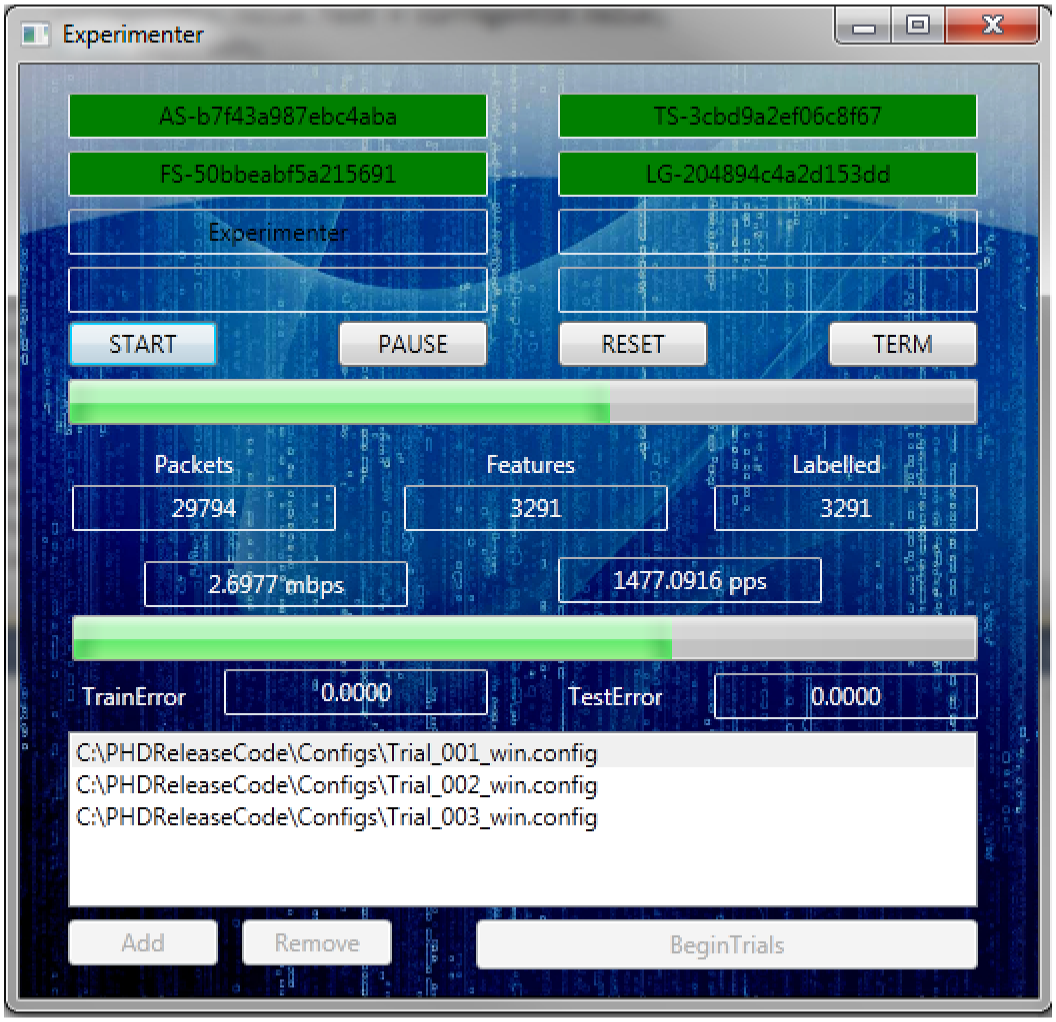
5.5.7 Observer Agent Implementations

A central feature of the Multi-Agent Malicious Behaviour Detection framework is interaction with network defenders. To validate this research I implemented three Observer Agents that enable human-machine interaction: *Logger Agent*, *Experimenter*, and *Network 3D User Interface*. Each of these agents serves a specific purpose with respect to human-machine interaction.

5.5.7.1 Logger Agent

The Logger Agent monitors all of the log messages, as well as many other messages produced by agents while the Multi-Agent Malicious Behaviour Detection system is running. The Logger Agent subscribes to the Feature, Labelled, AgentControl, and AgentLog exchanges. When agents write to any of these exchanges, the Logger Agent formats the message into an ASCII string and writes it to a file. Each agent will periodically publish a log message to the AgentLog exchange (Section 5.4.1). Each log message contains a time stamp, the agent's unique id, the agent's *control state* (ready, running, paused, terminated, or complete), and other agent-specific information. For example, Traffic Source Agents include the number of packets published, packets published per second, and the number of megabits of traffic published per second.

A human network defender (or other agents) can review the log files produced by the Logger Agent to support malicious multi-agent system behaviour analysis, can monitor the log file in real time when a graphical user interface is not available, or can use it to debug Multi-Agent Malicious Behaviour Detection Agents.



AS-b7f43a987ebc4aba
TS-3cbd9a2ef06c8f67
FS-50bbeabf5a215691
LG-204894c4a2d153dd
Experimenter

Packets: 29794
Features: 3291
Labelled: 3291

2.6977 mbps
1477.0916 pps

TrainError: 0.0000
TestError: 0.0000

```
C:\PHDReleaseCode\Configs\Trial_001_win.config
C:\PHDReleaseCode\Configs\Trial_002_win.config
C:\PHDReleaseCode\Configs\Trial_003_win.config
```

Figure 5.10: The main window of the experimenter.

5.5.7.2 Experimenter

The Experimenter provides network defenders a mechanism for running repeatable experiments with a set of agents, a predefined configuration, and pre-recorded network traffic. Figure 5.10 shows the user interface. The top quarter of the interface displays the unique IDs of agents participating in the trial. For example, AS-b7f43a987ebc4aba is an Alert Source Agent. The colour of the boxes containing the agent IDs represents the agent's current state: yellow for ready, green for running, blue for paused, red for terminated, and purple when done processing all input. Below the agent ID boxes are buttons to send various messages to the AgentControl exchange, to start a trial, pause a trial, reset a trial, or terminate a trial. If the number of packets for the trial is known beforehand, the first progress bar shows how many packets have been processed. Following that are boxes for various trial characteristics, including the number of packets processed, the number of feature sets processed, the number of feature sets labelled by an alert source agent, the bandwidth of the packet manager in megabits per second, and the bandwidth in packets per second. Next is a progress bar displaying the ratio of alerts received so far, if the total number of alerts in the trial is known. Below the second progress bar are the training and test errors of the Machine Learning Agents participating in the trial. Finally, there is a text box displaying a list of trial configurations and buttons to add, remove, and begin trials.

Each experimental trial depends on an xml configuration file that sets out the parameters of the trial. It indicates what agents are required for the trial, the directory for any agent-specific files, the AMQP server details, the location of the pre-recorded network traffic, the agent logging interval, and other minor settings. Once launched, the experimenter ensures that each agent that should be involved in the trial launches successfully, and periodically checks on the agents to ensure they are properly processing traffic. The experimenter uses the logging messages as a heartbeat and parses the information in the log files to determine the status of the agents.

The experimenter has been a valuable tool for validating the Multi-Agent Malicious Behaviour Detection system in Chapter 6. The complete documentation for the experimenter can be found in Appendix A.2.

5.5.7.3 Network 3D User Interface

While the Logger Agent enables network defenders to monitor agents, and the Experimenter enables some limited interaction between the Multi-Agent Malicious Behaviour Detection system and network defender for evaluating the autonomous parts of the system, the Network 3D User Interface is intended to be the operational interface for shared discovery of malicious multi-agent systems. The interface is a prototype, and I regularly modify the interface to account for new features. As such, it is complete for the purposes of this thesis, but will need ongoing modification as future work unfolds.

The Network 3D User Interface was written using Microsoft XNA Game Studio. It uses the DirectX 3D libraries to take advantage of the GPUs available on modern graphics cards to draw a 3D representation of the network the Multi-Agent Malicious Behaviour Detection system is deployed in. Figure 5.11 illustrates an example of this user interface displaying traffic.

Each machine in the network is drawn as a sphere. Spheres of the same colour represent machines belonging to the same logical network. Each sphere is labelled with an IP address and, if a Protocol Analysis Agent has identified a host name, the most recent host name associated with the IP address. Packets are drawn on the user interface using 2D *particles*. A particle is a small 2D image (jpeg, gif, png, or other) that can be drawn quickly into a 3D scene. Each particle represents 10 packets. As machines in the network interact with each other, it is visible on the user interface as

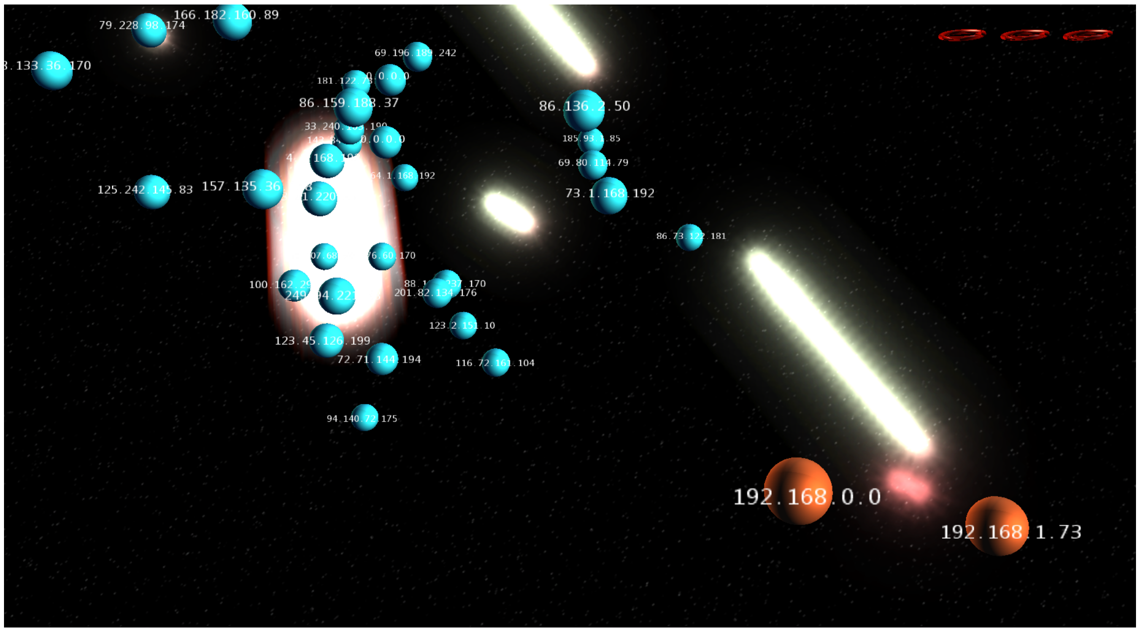


Figure 5.11: The main window of the Network 3D User Interface.

streams of particles between the spheres.

On the top right of the interface is a series of spinning tori. Each torus represents a Multi-Agent Malicious Behaviour Detection Agent. A torus performs a partial rotation each time the interface receives a message from an agent. The speed with which the torus rotates gives the network defender a rough idea of how much work each agent is performing. If a torus stops moving all together, the agent is either no longer running, or the agent is no longer publishing any messages to exchanges the interface is subscribed to.

Clicking on agents or machines opens a menu listing interactions that the network defender can take. For example, adding a machine to a whitelist, or querying a Protocol Analysis Agent for information about an IP address. The items in the menu depend on the type of agent, or the machine clicked on.

A window across the bottom of the interface displays notifications from the various agents, and can also be used to stream a dump of the network traffic or agent logs.

The network defender is also capable of navigating around the network by focusing on a single machine or free flying around the 3D environment. When focused on a single machine, the interface allows the network defender to orbit around and zoom in and out using the machine as a centre. When free flying, the network defender can move the focus up, down, left, right, in and out. The movement is similar to what one would expect in a flight simulator. The movement through the 3D interface enables the network defender to focus in on areas of interest in the network.

As mentioned earlier, this user interface represents ongoing work. Human-machine interaction is important to validating this research, and further improving this user interface is an important area of future research.

5.6 Summary

This chapter described many aspects dealing with an implementation of the Multi-Agent Malicious Behaviour Detection framework. At this point, those wishing to replicate or extend this system should have enough information to do so (in conjunction with the relevant appendices referred to throughout this chapter). The implementation here is used to validate the Multi-Agent Malicious Behaviour Detection design detailed in Chapter 4 in a series of experiments described in the next chapter. The series of experiments (and the analysis of the results of those experiments) are designed to evaluate the Multi-Agent Malicious Behaviour Detection approach and answer the research questions posed in Chapter 1.

Chapter 6

Evaluation

6.1 Overview

In order to validate the Multi-Agent Malicious Behaviour Detection framework implementation described in Chapter 5, I devised three sets of experiments. The first set is intended to provide a demonstration of the framework's capability of learning to classify benign traffic. This experiment evaluates a number of different machine learning algorithms in the process. The experiment also allows the system to demonstrate its capability of processing network traffic at a reasonable speed, and tests the cooperation of agents to achieve reasonable classifications of network traffic. The second set of experiments focuses on the human-machine discovery task, and uses datasets consisting of both malicious and benign traffic. The malicious traffic is woven into benign traffic at several benign-to-malicious ratios. This experiment allows the examination of how the proportion of malicious traffic impacts the system's ability to make effective recommendations to network defenders. The last set of experiments passes a full set of network traffic that emulates a higher education establishment's network traffic, with a large amount of various malicious attacks blended in. The final experiment approximates a real world network detection task, and the Multi-Agent Malicious Behaviour Detection implementation attempts to identify malicious traffic using the *classify only malicious* technique as described in Session 4.5.3.

In the next section I describe the shared portions of the methodology for all experiments. For each set of experiments I will outline the purpose, discuss experiment-specific methodology, provide descriptions of the network traffic (test datasets), and the results of the experiments. A discussion of the results will follow each experiment. Further analysis of these results will then follow in Chapter 7.

6.2 Experimental Environment

In the following subsections, I will describe three aspects of the experimental environment:

1. The physical machines that supported the experiments.

2. The types of agent implementations deployed for the experiments.

3. The network traffic processed in the experiments.

6.2.1 Physical Hardware

All experimental trals were performed with agents divided across three physical machines. Network connectivity was enabled by a commodity router providing a single class C network. The physical hardware consisted of a desktop PC (*Blackhole*),

	Blackhole	Reddwarf	Ionstorm	Nebula
Platform	WRT320N	Desktop PC	Mac Mini	Dell Precision M6400
OS	DD-WRT v24-sp2	Win 7 Pro SP1	OS X 10.7.2	Win7 Pro SP1
CPU	BCM4716 354 Mhz	Q6600 2.4 GHz	Core2 Duo 2.0 GHz	Q9300 2.54 GHz
Cores	1	4	2	4
RAM (GB)	0.03	4.00	2.00	8.00

Table 6.1: Hardware available for experiments.

a Mac Mini (*Reddwarf*), a Dell laptop (*Ionstorm*) and a Linksys router (*Darknebula*). The specifications for the hardware are described in Table 6.1.

6.2.2 Agents

Each experimental trial involved a subset of the following impemented agents, already described in Chapters 4 and 5:

- Packet Manager Agent
- Unified Alert Source Agent
- Machine Learning Agent
- Logger Agent
- Experimenter Agent

The Packet Manager Agent, Unified Alert Agent, Logger Agent and Experimenter Agent resided on Blackhole, while the Machine Learning Agents resided on Reddwarf. Ionstorm hosted a dedicated RabbitMQ AMQP server version 2.7.1 to handle all inter-agent communication.

6.2.3 Network Traffic

Traffic for all experimental trials was generated using a *Breaking Point* appliance [Wright et al., 2010; Tarala, 2011]. The Breaking Point appliance generates network traffic to simulate a variety of network traffic loads, simulated users, and network protocols [Beyene et al., 2012]. The Breaking Point appliance is designed specifically to stress test network security solutions by introducing malicious traffic into the generated network traffic. The Breaking Point enables the seamless introduction of malware traffic through *Strike Packs*, which are intended to weave malicious attacks into the benign network traffic. One such appliance is capable of generating 120 Gpbs of simulated traffic to test network infrastructure, such as firewalls, intrusion detection system, web servers, and routers. The platform has been used by a number of private companies, banks, and telecoms including Yahoo, EUCOM, Korea Telcom, the University of Wisconsin, Cisco, Northrop Grumman UK, and Juniper. Case studies for all of the previous mentioned institutions are available on the BreakingPoint website [BreakingPoint Systems, 2012].

I generated four distinct sets of traffic for validating the Multi-Agent Malicious Behaviour Detection implementation, with the goal of providing consistent traffic types to validate the system's capability to identify both malicious and benign sessions.

6.2.3.1 Benign Traffic Set 1

Benign Traffics Set 1 consists of 1.06 gigabytes of network traffic with the following breakdown of application protocol types: 7.02% Telnet, 68.21% generic HTTP, 3.78% Yahoo Mail, 2.85% SSH, 9.71% Gmail, 4.72% DNS, and 3.71% NNTP. All of the

Benign Traffic Set 1 packets were IP over Ethernet, with a total of 5801533 packets. At the transport layer the traffic is made up of 5282291 TCP packets and 519242 UDP packets. The remaining packets were ignored, and consisted of mostly ICMP.

Benign Traffic Set 1 was generated with a limited set of protocols. While none of the protocols are specifically attributed to multi-agent systems per se, they are protocols that are often exploited by malicious multi-agent systems, and they are popular protocols one would expect to be seen used by a variety of users in a typical network environment. These non-malicious protocols share some of the same behaviours one would expect from malicious multi-agent systems, such as beaconing (DNS and NNTP), exfiltrating (Gmail and Yahoo Mail), and updating (HTTP). Classifying simple benign traffic that shares some of the same behaviours as malicious multi-agent systems is an important first step in validating the Multi-Agent Malicious Behaviour Detection system to show that the traffic can be later whitelisted if required.

6.2.3.2 Benign Traffic Set 2

Benign Traffic Set 2 consists of 1.05 gigabytes of network traffic with the following breakdown of application protocol types: 5.32% Telnet, 53.20% generic HTTP, 1.76% Yahoo Mail, 1.97% SSH, 7.53% Gmail, 3.85% DNS, 3.75% Oscar IM, 1.69% GTalk, 4.98% IRC, 0.2% BitTorrent Tracker, 1.69% BitTorrent Peer, 0.31% Facebook, 1.21% Netflix streaming video, 2.30% OSCAR file transfer, 6.44% eDonkey, and 4.09% NNTP. All of the Benign Traffic Set 2 packets were IP over Ethernet, with a total of 5735045 packets. At the transport layer the traffic is made up of 5239869 TCP packets and 1097331 UDP packets. The remaining packets were ignored, and consisted of mostly ICMP.

Benign Traffic Set 2 has the same protocols as Benign Traffic Set 1, with the addition of some benign multi-agent system traffic. Peer-to-peer traffic, such as eDonkey and BitTorrent, represent multi-agent systems that beacon to trackers to receive locations for peers and then download chunks of files (update), and share information from the local machine (exfiltrate). Effectively, a bittorent network performs all of the tasks that a typical malicious multi-agent system does. Identiifying such networks supports the validation of the Multi-Agent Malicious Behaviour Detection implementation. Chat protocols are often abused by malicious multi-agent systems for communicating command and control. For that reason I included some OSCAR, GTalk, and IRC traffic in the sample in small amounts to mimic malicious multi-agent system communications. Social networks, such a Facebook, support a number of functions that are comparable to what one might expect from a multi-agent system.

6.2.3.3 Benign Traffic Set 3

Benign Traffic Set 3 consists of 2.12 gigabytes of network traffic with the following breakdown of application protocol types: 4.77% Telnet, 48.15% generic HTTP, 3.09% Yahoo Mail, 2.59% SSH, 6.52% Gmail, 3.30% DNS, 3.57% Oscar IM, 3.55% GTalk, 5.04% IRC, 0.2% BitTorrent Tracker, 3.63% BitTorrent Peer, 0.24% Facebook, 2.70% Netflix streaming video, 2.34% OSCAR file transfer, 5.91% eDonkey, and 4.61% NNTP. All of the Benign Traffic Set 2 packets were IP over Ethernet, with a total of 5735045 packets. At the transport layer the traffic is made up of 5239869 TCP packets and 1097331 UDP packets. The remaining packets were ignored, and consisted of mostly ICMP.

Benign Traffic Set 3 is a larger version of Benign Traffic Set 2. It was generated with the same general profile, and is included in order to ensure that the results of trials on Benign Traffic Set 2 can be repeated on a second similar dataset.

6.2.3.4 Malicious Traffic Set

The Malicious Traffic Set was generated by using a Breaking Point network profile in combination with a series of strike packs. The *Higher Education* network profile generates a variety of application level traffic typical of a college or university campus network. In addition to the default profile, I added a few additional chat protocols and online social network protocols to mimic the behaviours one might find in a multi-agent system. Malicious traffic was woven into the Malicious Traffic Set by the Breaking Point appliance. The Malicious Traffic Set contains a total of 3.4 gigabytes of traffic.

The traffic contains a variety of benign multi-agent traffic, as in the first three data sets. The volume of malicious attacks introduced is intended to represent malicious multi-agent systems propagating through various exploits, updating by retrieving malicious documents, beaconing through DNS lookups to malicious domains, denial of service attacks, and various attacks that mimic a number of protocols. The volume is important to match the characteristics of malicious multi-agent systems. The goal of this data set was to provide a rich set of malicious traffic that could be manipulated to represent malicious multi-agent system communications. The traffic set is relatively safe, and could be passed through the system with relatively little risk of infecting machines involved in the experiments.

6.3 Benign Traffic Experiments

6.3.1 Purpose

The first set of experiments is intended to demonstrate my framework's capability of performing the passive tasks described throughout this thesis; that is: read network traffic, distribute the features of the traffic to other agents, and label traffic based on the derived features. Additionally, the experiments are intended to validate the claim that an online machine learning algorithm is capable of classifying both benign multi-agent and single agent traffic based on provided features, and achieve accuracy similar to a signature-based misuse detection system. The experiments are also intended to show that the architecture can perform the task described fast enough to interact with live traffic, validating that the architecture is usable for traffic manipulation as described in the Chapter 4.

6.3.2 Methodology

In order to test the accuracy of a variety of machine learning algorithms, a total of five machine learning agents were selected based on the algorithm implementations provided in Saffari et al. [2010]. The algorithms are Online Random Tree, Online Random Forest, Online LaRank, Online Multi-Class Gradient Boost, and Online Multi-Class Linear Programming Boost (Sections 3.3.6, 4.5.3, and 5.5.3).

The Multi-Agent Malicious Behaviour Detection implementation was trialled on

Benign Traffic Sets 1, 2, and 3. The benign traffic experiments consist of a total of 60 trials, 20 trials for each of the three benign data sets. For each of the three data samples, one of four testing vs. training ratios was used: 1:9, 1:3, 1:1, and 3:1. The different ratios are intended to simulate the Alert Source Agents' capability to label network traffic. Higher testing ratios simulate poorer performance on the part of an Alert Source Agent and higher demands on Machine Learning Agents to classify unlabelled traffic, given the smaller ratio of training samples available for learning.

While the exact individual trial set-ups were not repeated, by repeating the same machine learning algorithms on different testing vs. training ratios with the same data sets, I hypothesized that the results would show a gradual change in performance. Further, by repeating the machine learning and testing vs. training ratios on different, but somewhat similar data sets, over a total span of 60 trials, any equipment failures or anomalies in the performance of the Multi-Agent Malicious Behaviour Detection implementation would be obvious in the results.

In order to generate labels for the traffic, I wrote Snort signatures capable of identifying each of the traffic types in the sample. In each trial Snort processes the traffic sample with a set of Snort signatures to generate a Unified Alert log. The Experimenter Agent then sends messages out to each individual agent on the agent control channel announcing a trial has commenced. The Packet Manager begins processing packets from the sample traffic file at a rate of between 20-50 mbps. The speed was chosen specifically to represent a small business, or modern residential internet connection. Given the modest hardware available for these experiments, a residential or home network speed was deemed a reasonable goal. The Unified

Alert Agent begins processing the Unified Alerts and publishing them to an alerts exchange for the other agents. The Packet Manager Agent passes traffic up to one of two Transport Layer Traffic Sources, one for UDP and one for TCP. The Transport Layer Traffic Sources derive features from the traffic and check for labels from the Unified Alert Source, labelling feature sets if labels are available. Then the Transport Layer Traffic Source Agents publish the labelled traffic to the features exchange. Machine Learning Agents receive the features, determine whether to test or train on each sample feature set, and broadcast the results to a Labelled traffic exchange. Periodically, each agent publishes a log message, and the Logger Agent collects all the log messages and writes them to disk.

UDP features are automatically timed out by the Transport Layer Feature Sources after a fixed number of milliseconds (700) of no observed packets matching the session. 700 milliseconds was chosen heuristically. Smaller values tended to prematurely end several UDP sessions, while larger values added unnecessary delay. TCP features are only published if a complete TCP close is observed in traffic, requiring both a client packet with the TCP FIN flag set and a server packet with the TCP FIN flag set.

6.3.3 Performance Evaluation

The results of each trial were measured against three criteria.

First, the accuracy of the Machine Learning Agents was measured across the trials for each training and testing ratio and compared with the other Machine Learning Agents. Two measures of accuracy were considered: the overall accuracy across all available testing samples, and a periodic measurement of accuracy over subsets of 1% of the tested samples. The periodic accuracy measurements account for the online machine learning algorithms' capability of continual adaptation over time. The standard deviation across the 100 1% intervals is also compared as an indicator of the fluctuation of the Machine Learning Agents' performance.

Second, I examine what feature sets provided the most difficulty for the classification process, with the goal of validating the systems capability of identifying unusual traffic. The traffic is intended to simulate real world network traffic, where protocols are rarely evenly distributed. For example, the typical network might contain ten times more HTTP traffic than SSH traffic. By looking at the classification accuracy at the individual protocol level, I can determine if the Machine Learning Agents are capable of learning to identify traffic with few samples, as well as resisting a bias to traffic with large sample counts.

Finally, I measure the time taken from the moment a feature is published to the moment the Machine Learning agent classifies the feature set. The measurement depends on the time stamps provided by the Logger Agent, as it will write out each feature set published by the Packet Manager, and each labelled feature set published by the Machine Learning Agent. By comparing the mean, standard deviation, minimum and maximum delay, I can demonstrate the system's capability to react to traffic quickly enough to manipulate traffic in a meaningful fashion.

The next three sections describe the data samples and the results of the trials run on each sample.

6.3.4 Benign Traffic Set 1

Trials 1 to 20 involved passing Benign Traffic Set 1 through a Snort process with the following Snort rules developed to label the traffic:

alert tcp any 80 <>any any (content: “yahoo” , nocase; msg: “YahooMail”; sid: 1;)

alert tcp any 80 <>any any (content: “Cookie:”; content: “gmail”, nocase; within: 100; msg: “GMail”; sid: 2;)

alert tcp any 80 <>any any (flags: SA; msg: “HTTP”; sid: 3;)

alert udp any 53 <>any any (msg: “DNS”; sid: 4;)

alert tcp any 22 <>any any (flags: SA; msg: “SSH”; sid: 5;)

alert tcp any 23 <>any any (flags: SA; msg: “TELNET”; sid: 6;)

alert tcp any 119 <>any any (flags: SA; msg: “NNTP”; sid: 7;)

6.3.5 Results

The charts in Figures 6.1 to 6.4 show the accuracy of each of the Machine Learning Agents on Benign Traffic Set 1, where each point is the accuracy achieved over the last 1% of samples tested. All of the Machine Learning Agents perform well at the start of the test, as the clients in the network begin by making a variety of DNS requests in preparation for connecting to various services. In each chart there is a significant dip in performance after 20% of the traffic has been sampled. This demonstrates the algorithms adjusting to the variety of additional protocols in the traffic. Note that some of the algorithms take a significant amount of time to regain their accuracy. In particular, the Online Random Tree consistently performs poorly throughout the

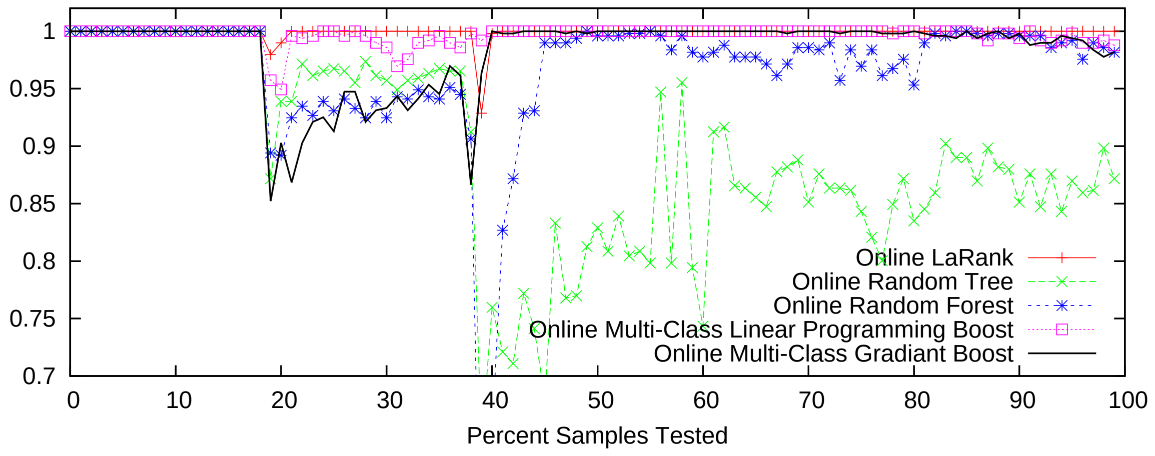


Figure 6.1: Accuracy results on Benign Traffic Set 1. Trained on 90% and Tested on 10%. The y-axis represents the test accuracy.

trials. It is also worth noting that as the training to testing ratios shift toward fewer training samples, most of the Machine Learning Agents demonstrate dips in performance. For example, there is a dip in accuracy of the Online Multi-Class Linear Programming Boost Agent just after 50% of the samples have been processed in the 1:1 and 1:3 ratio charts. The charts indicate that while Online Multi-Class Gradient Boost, Online LaRank and Online Random Forest are the most resistant to changes in the traffic texture and the training to testing ratio, Online LaRank appears to have the most robust accuracy.

Table 6.2 shows the accuracy and standard deviation across the trials at the various ratios, and includes the 99$^{\text{th}}$, 75$^{\text{th}}$, and 50$^{\text{th}}$ sample accuracy. The values are very telling as to how each of the Machine Learning Agents are capable of maintaining a consistent accuracy. The LaRank Agent has both a high accuracy and a consistent standard deviation, while Online Multi-Class Linear Programming Boost shows an increase both standard deviation and a decrease in performance. The LaRank Agent

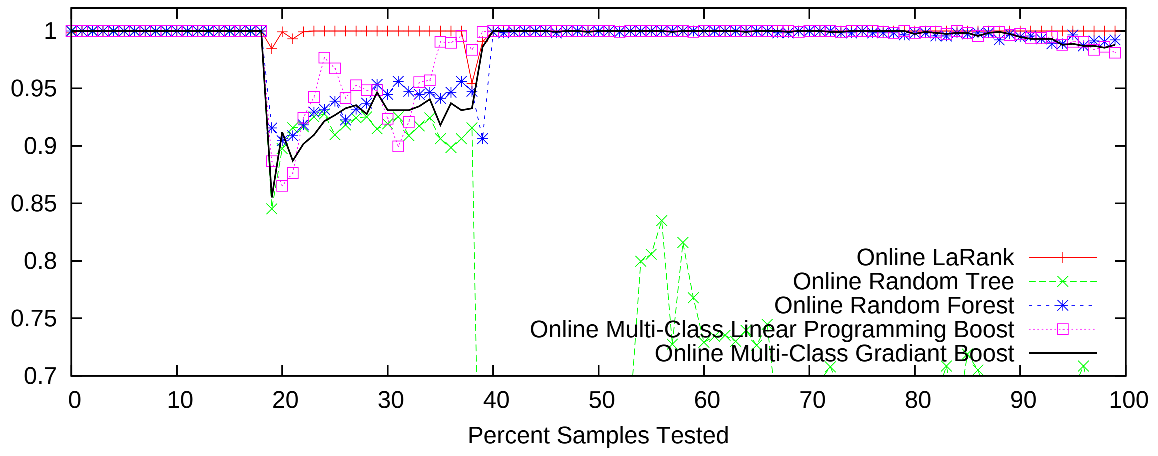


Figure 6.2: Accuracy results on Benign Traffic Set 1. Trained on 75% and Tested on 25%. The y-axis represents the test accuracy.

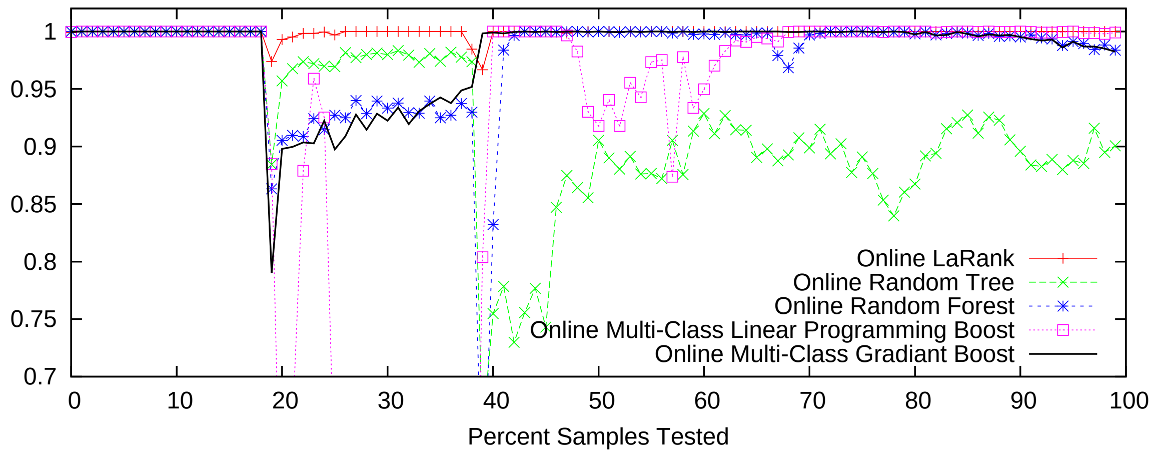


Figure 6.3: Accuracy results on Benign Traffic Set 1. Trained on 50% and Tested on 50%. The y-axis represents the test accuracy.

appears to perform the best on Benign Traffic Set 1. However, according to accuracy and standard deviation both the Online Multi-Class Gradient Boost and Online Random Forest are competitive.

Table 6.3 breaks down the classification accuracy across the different protocols in the traffic. Given that individual samples were randomly chosen for the purposes of testing or training while maintaining the target ratio, the mean samples column is

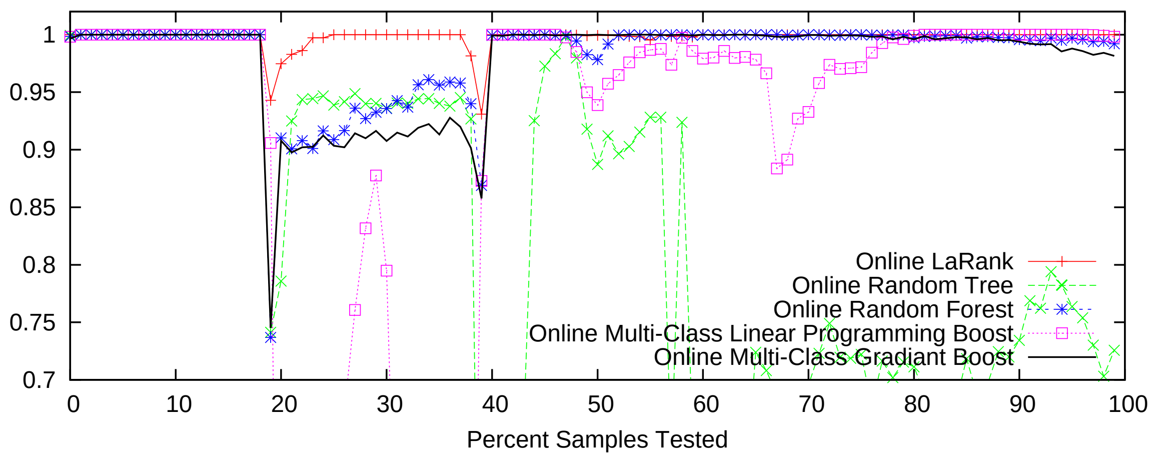


Figure 6.4: Accuracy results on Benign Traffic Set 1. Trained on 25% and Tested on 75%. The y-axis represents the test accuracy.

9:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.8922	0.9655	0.9965	0.9828	0.9990
Stdv	0.0427	0.0189	0.0019	0.0120	0.0007
99th pct	0.8982	0.9857	0.9919	0.9777	1.0000
75th pct	0.8635	0.9837	1.0000	1.0000	1.0000
50th pct	0.8106	1.0000	1.0000	0.9980	1.0000

1:3	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.7700	0.9850	0.9872	0.9830	0.9992
Stdv	0.0884	0.0104	0.0094	0.0121	0.0055
99th pct	0.6836	0.9905	0.9863	0.9853	1.0000
75th pct	0.6707	0.9983	1.0000	1.0000	1.0000
50th pct	0.5795	0.9991	1.0000	0.9991	1.0000

1:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9182	0.9763	0.8860	0.9817	0.9990
Stdv	0.0323	0.0151	0.0818	0.0135	0.0040
99th pct	0.8949	0.9882	0.9978	0.9851	1.0000
75th pct	0.8778	1.0000	1.0000	1.0000	1.0000
50th pct	0.8560	0.9996	0.9301	0.9996	1.0000

3:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.8257	0.9818	0.8784	0.9772	0.9978
Stdv	0.0604	0.0139	0.0845	0.0172	0.0016
99th pct	0.7031	0.9941	0.9991	0.9842	1.0000
75th pct	0.7184	1.0000	0.9709	0.9994	1.0000
50th pct	0.9179	0.9826	0.9500	0.9997	0.9991

Table 6.2: Comparison of the accuracy of the five Machine Learning Agents when trained on Benign Traffic Set 1 at various training:testing ratios.

provided to show how many samples the algorithms encountered on average of each protocol for testing. The results help to identify where specific agents broke down in their capability to identify specific protocols. Note that across all of the trials, the LaRank agent is the only Machine Learning Agent capable of distinguishing Yahoo Mail traffic from the other protocols. The LaRank agent's success in distinguishing Yahoo Mail from other protocols, when few Yahoo Mail samples are available, demonstrates the LaRank algorithm's capability to learn to classify traffic with few training samples present. Most of the Agents, with the exception of LaRank, were very poor at identifying SSH, and only Online Multi-Class Linear Programming Boost could compete with LaRank with respect to the classification of Telnet traffic. The performance on classification of SSH traffic may have been impacted by encryption, as many encrypted and compressed protocols have similar features (Section 3.8). DNS and HTTP made up the most feature sets, and each of the Agents could identify the traffic with relatively good accuracy. Consider that while Online Multi-Class Gradient Boost performed with a very high accuracy across all samples, it performed poorly against protocols with limited sample availability.

Table 6.4 shows the latency. The latency on Benign Traffic Set 1 does not appear to show any specific pattern, as the ratio of training to testing samples changes. However, it should be noted that the standard deviation across all trials varies wildly. Also, there are a number of instances where the minimum latency is negative, indicating strange behaviour on the part of the Logger Agent. The negative values will be further discussed in section 6.3.11. The latency with the Online Random Tree and Online Random Forest have relatively higher maximum latency than the other

9:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	4	0.0000	0.0000	0.0000	0.0000	**1.0000**
Gmail	560	0.8545	0.8998	0.9891	0.8602	**0.9982**
HTTP	19889	0.9444	0.9840	0.9961	0.9980	**0.9988**
DNS	28064	0.8658	0.9757	**1.0000**	0.9977	0.9991
SSH	43	0.0000	0.0000	0.6154	0.0000	**0.9697**
Telnet	671	0.5446	0.1074	0.9052	0.0772	**0.9971**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000

1:3	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	11	0.0000	0.0000	0.0000	0.0000	**0.8750**
Gmail	1321	0.4097	0.8324	0.9948	0.8438	**0.9977**
HTTP	46925	0.7606	0.9987	0.9738	0.9979	**0.9995**
DNS	66337	0.8035	0.9984	**0.9998**	0.9995	0.9990
SSH	107	0.0000	0.0000	0.6724	0.0000	**0.9891**
Telnet	1547	0.0000	0.2097	0.8795	0.0085	**0.9993**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000

1:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	22	0.0000	0.0000	0.0000	0.0000	**0.7037**
Gmail	2592	0.9153	0.8483	0.9934	0.7627	**0.9985**
HTTP	92339	0.9533	0.9958	0.7205	0.9963	**0.9989**
DNS	130309	0.8998	0.9891	0.9990	**0.9998**	0.9994
SSH	215	0.0000	0.0000	0.8111	0.0000	**0.9427**
Telnet	3040	0.7054	0.0171	0.9864	0.0462	**0.9966**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000

3:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	34	0.0000	0.0000	0.0000	0.0000	**0.9412**
Gmail	3880	0.2061	0.7719	0.9905	0.6097	**0.9959**
HTTP	137188	0.7101	0.9935	0.7014	0.9950	**0.9972**
DNS	194209	0.9272	0.9973	**0.9997**	0.9970	0.9984
SSH	319	0.0863	0.0000	0.5382	0.0000	**0.9719**
Telnet	4500	0.5358	0.2074	0.9835	0.0000	**0.9922**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000

Table 6.3: Comparison of the accuracy achieved by each Machine Learning Agent when trained on 90% of samples and tested on 10% of samples for Benign Traffic Set 1. Also an indicator of how many samples for each protocol the agent was tested on.

Machine Learning Agents. In this case, the mean latency is likely the best indicator of the performance of the agents, as it gives at least some indication of whether an agent has any chance of receiving a message in time to manipulate traffic. With the more accurate Machine Learning Agents, the maximum latency was between 1 and 6

Latency (9:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	7315.8328	1235.3161	338.1691	695.4499	913.1492
Stdv	9153.3032	1651.4664	617.3855	1000.2255	1251.8602
Min	-9.0000	-35.0000	-19.0000	-19.0000	-17.0000
Max	29076.0000	4895.0000	2886.0000	3878.0000	4154.0000

Latency (3:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	3705.6160	2633.5824	184.8015	273.7871	1386.2404
Stdv	5335.6552	3771.8771	283.2198	385.3007	1936.6638
Min	-10.0000	-11.0000	-68.0000	-39.0000	-19.0000
Max	16308.0000	11788.0000	1315.0000	1778.0000	6161.0000

Latency (1:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	764.1758	1274.3003	1233.6562	1275.0440	653.9160
Stdv	1050.1528	1665.4100	1706.3531	1707.4604	941.2019
Min	-33.0000	-12.0000	-91.0000	-83.0000	-8.0000
Max	3456.0000	5143.0000	5290.0000	5585.0000	3414.0000

Latency (1:3)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	1877.3860	2797.5988	278.6633	611.3509	1016.9046
Stdv	2698.3398	3987.3828	425.2116	869.7252	1370.3494
Min	-30.0000	-280.0000	-20.0000	-44.0000	-41.0000
Max	8162.0000	11234.0000	2499.0000	3772.0000	4504.0000

Table 6.4: Comparison of the latency of the system from the time the Logger Agent receives a feature to the time the Logger Agent receives an associated label from the Machine Learning Agent, when trained on 90% of samples and tested on 10% of samples for Benign Traffic Set 1. All latency values are in milliseconds.

seconds, which is not fast enough to manipulate traffic in a meaningful way. However, the mean values show some promise, as values between 0 to 300 milliseconds are more reasonable. I will return to these results in section 6.3.11.

6.3.6 Benign Traffic Set 2

Trials 21 to 40 processed the Benign Traffic Set 2, and used the following Snort rules to initially label the traffic:

alert tcp any 80 <>any any (content: "yahoo", nocase; msg: "YahooMail"; sid: 1;)

alert tcp any 80 <>any any (content: “Cookie:”; content: “gmail”, nocase; within: 100; msg: “GMail”; sid: 2;)

alert tcp any 80 <>any any (content: “GET /announce?peer_id=”; msg: “BitTorrent Tracker”; sid: 3;)

alert tcp any 80 <>any any (content: “facebook”, nocase; msg: “Facebook”; sid: 4;)

alert tcp any 80 <>any any (content: “netflix”, nocase; msg: “Netflix”; sid :5;)

alert tcp any 80 <>any any (flags: SA; msg: “HTTP”; sid: 6;)

alert udp any 53 <>any any (msg: “DNS”; sid: 7;)

alert tcp any 22 <>any any (flags: SA; msg: “SSH”; sid: 8;)

alert tcp any 23 <>any any (flags: SA; msg: “TELNET”; sid: 9;)

alert tcp any 119 <>any any (flags: SA; msg: “NNTP”; sid: 10;)

alert tcp any any <>any ![80,53] (content: “BitTorrent protocol”, nocase; msg: “BitTorrent Peer”; sid: 11;)

alert tcp any any <>any any (pcre: “/[A-Za-z]{12}/”; content: “—00—1.0”; within: 10; msg: “eDonkey”; sid: 12;)

alert tcp any any <>any any (content: “OFT2”; content: “Cool FileXfer”; within: 100; msg: “Oscar File Transfer”; sid: 13;)

alert tcp any 5190 <>any ![80,53] (msg: “OSCAR”; sid: 14;)

alert tcp any 5222 <>any ![80,53] (msg: “GTALK”; sid: 15;)

alert tcp any 6667 <>any ![80,53] (msg: “irc”; sid: 16;)

alert tcp any 443 <>any any (content: “netflix”, nocase; msg: “Netflix”;

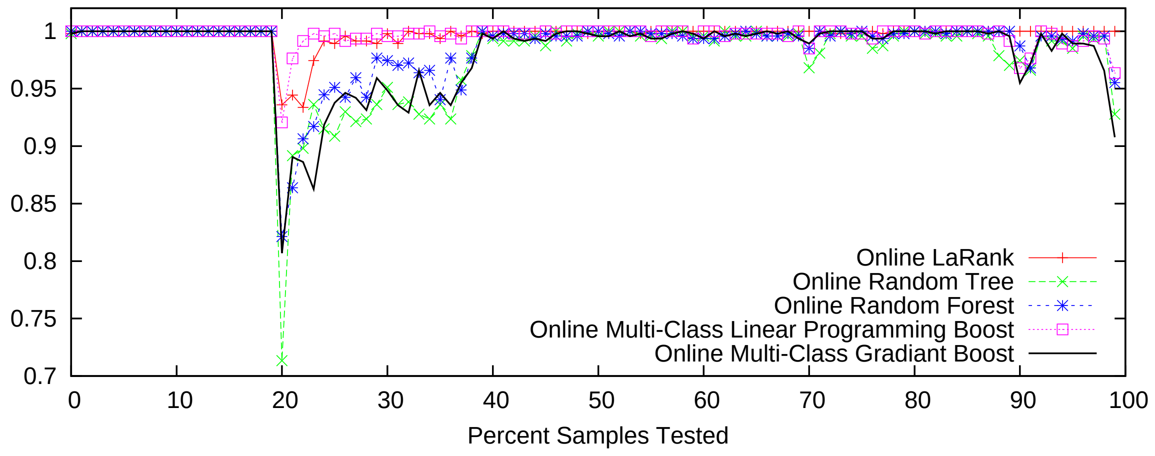


Figure 6.5: Accuracy results on Benign Traffic Set 2. Trained on 90% and Tested on 10%. The y-axis represents the test accuracy.

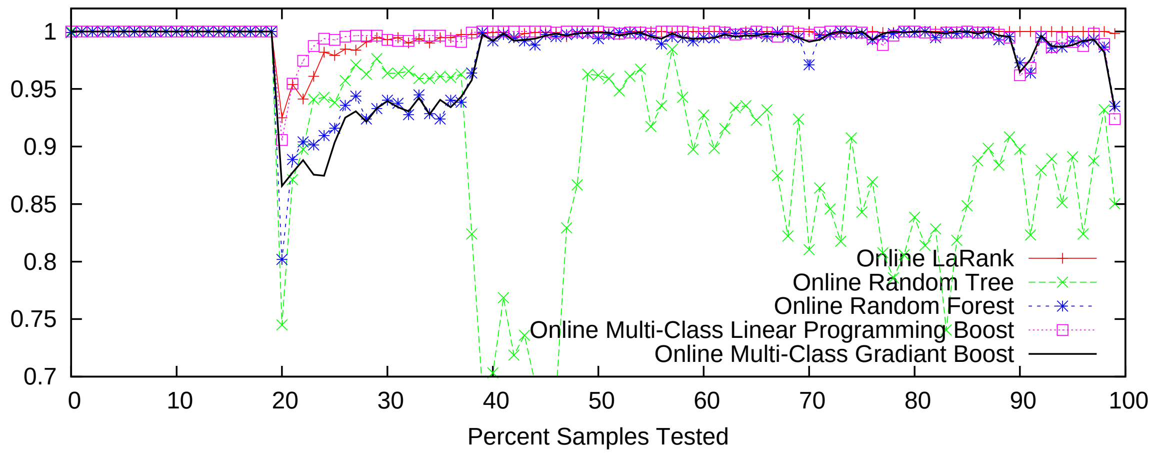


Figure 6.6: Accuracy results on Benign Traffic Set 2. Trained on 75% and Tested on 25%. The y-axis represents the test accuracy.

sid: 17;)

6.3.7 Results

The charts in figures 6.5 to 6.8 are similar to the results in figures 6.1 to 6.4. Each of the Machine Learning Agents achieves 100% as the traffic starts out, but as more

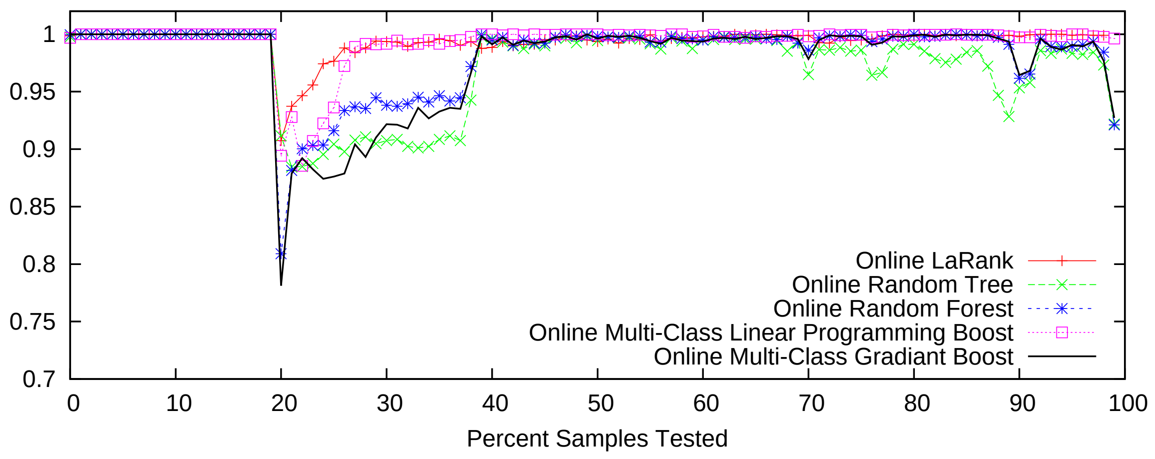


Figure 6.7: Accuracy results on Benign Traffic Set 2. Trained on 50% and Tested on 50%. The y-axis represents the test accuracy.

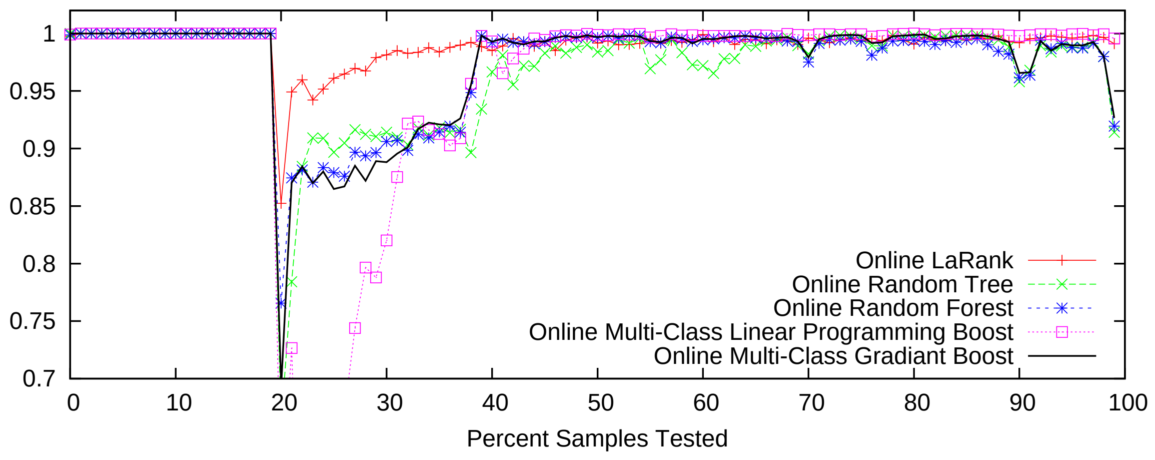


Figure 6.8: Accuracy results on Benign Traffic Set 2. Trained on 25% and Tested on 75%. The y-axis represents the test accuracy.

protocols are introduced, the performance of all algorithms suffers after 20% of the samples are processed. However, after 40% of the trials, all of the algorithms appear to settle back to to a relatively high accuracy compared to the period between 20% and 40% of samples. Even the Online Random Tree appears to perform much better with an increase in the variety of protocols available in the traffic to label. As with the first dataset, the Online LaRank Agent outperforms the other Agents, especially

9:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9797	0.9866	0.9961	0.9819	0.9969
Stdv	0.0135	0.0089	0.0016	0.0124	0.0064
99th pct	0.9958	0.9957	0.9936	0.9657	1.0000
75th pct	0.9958	0.9979	1.0000	1.0000	0.9979
50th pct	1.0000	1.0000	1.0000	0.9979	0.9979

3:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9017	0.9809	0.9947	0.9808	0.9960
Stdv	0.0360	0.0124	0.0023	0.0125	0.0049
99th pct	0.9319	0.9864	0.9891	0.9820	1.0000
75th pct	0.9074	0.9982	1.0000	0.9982	0.9991
50th pct	0.9628	0.9982	1.0000	0.9991	0.9982

1:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9723	0.9818	0.9929	0.9774	0.9941
Stdv	0.0156	0.0118	0.0096	0.0148	0.0062
99th pct	0.9732	0.9843	0.9977	0.9770	0.9991
75th pct	0.9852	0.9995	0.9995	0.9986	0.9945
50th pct	0.9981	0.9991	1.0000	1.0000	0.9949

1:3	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9696	0.9737	0.9526	0.9738	0.9899
Stdv	0.0177	0.0168	0.0398	0.0175	0.0087
99th pct	0.9799	0.9799	0.9994	0.9802	0.9966
75th pct	0.9975	0.9950	0.9994	0.9988	0.9963
50th pct	0.9901	0.9978	0.9997	0.9985	0.9969

Table 6.5: Comparison of the accuracy of the five Machine Learning Agents when trained on Benign Traffic Set 2 at various training:testing ratios.

near the end of the trial when it appears as though all the other drop in accuracy again.

Table 6.5 shows various values for each trial's Machine Learning Agent classification accuracy. With the exception of the Online Random Tree, when the training set is large the Machine Learning Agents operate at close to 100% accuracy, but as the training set get smaller, each Agent's performance degrades, with Online Multi-Class Linear Programming degrading the most of the four highest performing agents. The LaRank agent continues to maintain the lowest standard deviation, which is very encouraging given the high accuracy of the results. The Online Random Tree per-

9:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	0	0.0000	0.0000	0.0000	0.0000	0.0000
Gmail	532	**0.9981**	0.8583	0.9942	0.8901	0.9947
BitTorrent Track	14	0.0000	0.0000	0.0000	0.0000	**1.0000**
FaceBook	1	0.0000	0.0000	0.0000	0.0000	0.0000
Netflix Stream	9	0.0000	0.0000	0.0833	0.0000	**0.8750**
HTTP	18116	0.9944	**0.9985**	0.9984	0.9979	0.9958
DNS	26775	0.9974	0.9993	**1.0000**	**1.0000**	0.9994
SSH	14	0.0625	0.0000	0.0769	0.0000	**0.8947**
Telnet	628	0.1646	0.3565	0.8619	0.0571	**0.9968**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	214	0.6550	**0.9579**	0.9292	0.9194	0.9234
Oscar File	0	0.0000	0.0000	0.0000	0.0000	0.0000
Oscar IM	460	0.8137	0.9350	**0.9891**	0.8656	0.9673
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	103	0.4643	0.7228	**0.9626**	0.7476	0.8913
Netflix Auth	2	0.0000	0.0000	0.0000	0.0000	**0.5000**

3:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	1	0.0000	0.0000	0.0000	0.0000	0.0000
Gmail	1218	0.9894	0.8143	0.9886	0.8284	**0.9968**
BitTorrent Track	38	0.0000	0.0000	0.0000	0.0000	**0.9189**
FaceBook	4	0.0000	0.0000	0.0000	0.0000	**0.7500**
Netflix Stream	22	0.0000	0.0000	0.0000	0.0000	**0.8571**
HTTP	42703	0.8531	**0.9984**	0.9976	0.9983	0.9940
DNS	63165	0.9458	0.9993	**1.0000**	**1.0000**	0.9990
SSH	34	0.0789	0.0000	0.0000	0.0000	**0.9259**
Telnet	1538	0.4404	0.0303	0.8058	0.0000	**0.9968**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	495	0.8580	0.9038	**0.9412**	0.9306	0.9186
Oscar File	0	0.0000	0.0000	0.0000	0.0000	0.0000
Oscar IM	1076	0.9573	0.9146	**0.9869**	0.8692	0.9587
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	235	0.6724	0.8650	**0.9512**	0.5940	0.9348
Netflix Auth	7	0.0000	0.0000	0.0000	0.0000	**1.0000**

Table 6.6: Comparison of the accuracy achieved by each Machine Learning Agent when trained and tested with a 9:1 and 3:1 ratio of samples for Benign Traffic Set 2. Also an indicator of how many samples for each protocol the agent was tested on.

formed significantly worse than the other four agents. The difference in performance between the Online Random Forest and the Online Multi-Class Gradient Boost is not significant with respect to overall classification accuracy.

Table 6.6 and Table 6.7 provide a much more detailed look at exactly how well each of the agents performs against the available protocols. The very first thing that stands out in these tables is that the uneven distribution of the traffic makes a significant impact on each Machine Learning Agent's capability to classify the traffic. In many cases, while the protocol existed in the traffic, there was not enough samples for the Machine Learning Agents to distinguish between the underlying protocol and the encompassing protocol. For example, when the training to testing ratio was 1:3, LaRank consistently classified the 9 instances of Facebook traffic as HTTP. Note that during that particular trial there were only 5 training samples of Facebook. However, when the training to testing ratio was 1:1, LaRank correctly classified three instances of Facebook as Facebook and the fourth as HTTP, after training on only 9 samples. Yahoo Mail also proved to be problematic, as there were not enough training samples for any of the Agents to successfully distinguish them from other traffic. However, for the purposes of these trials, it is clear that the Machine Learning Agents - and specifically the LaRank-based agent - are capable of classifying traffic even when only a relatively small number of samples are available, such as the case with Facebook, where a 75% accuracy was achievable with only 13 out of 429405 Facebook samples, and the one error identified the encompassing protocol.

Next, consider the latency of the system as described in Table 6.8. The latency of the system is somewhat more stable on the second dataset. It is clear from the table that there is a difference in the latency, whereby both the Online Random Tree and the Online Random Forest based Agents introduce more latency overall than the other three Machine Learning Agents. The standard deviation of the delays is still fairly high. However, it appears as though the Online Multi-Class Gradient Boost Machine Learning Agent is the fastest to provide feedback from a feature once it is

1:1	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	3	0.0000	0.0000	0.0000	0.0000	**0.2500**
Gmail	2403	0.4820	0.8904	0.9857	0.8233	**0.9962**
BitTorrent Track	85	0.0000	0.0000	0.0000	0.0000	**0.7553**
FaceBook	7	0.0000	0.0000	0.0000	0.0000	**0.7143**
Netflix Stream	40	0.0000	0.0286	0.0000	0.0000	**0.9545**
HTTP	83859	**0.9989**	0.9978	0.9870	0.9972	0.9927
DNS	123819	**1.0000**	0.9999	0.9999	1.0000	0.9969
SSH	68	0.0000	0.0000	0.0000	0.0000	**0.9630**
Telnet	2973	0.0269	0.0506	0.9817	0.0014	**0.9923**
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	989	**0.9263**	0.8890	0.9244	0.8938	0.9249
Oscar File	0	0.0000	0.0000	0.0000	0.0000	0.0000
Oscar IM	2092	0.3971	0.9180	**0.9898**	0.6899	0.9625
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	472	0.4594	0.6017	**0.9386**	0.3612	0.8562
Netflix Auth	11	0.0000	0.0000	0.0000	0.0000	**0.7778**

1:3	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	5	0.0000	0.0000	0.0000	0.0000	0.0000
Gmail	3573	0.8932	0.5842	0.9885	0.6531	**0.9894**
BitTorrent Track	128	0.4122	0.0000	0.0000	0.0000	**0.4839**
FaceBook	10	0.0000	0.0000	0.0000	0.0000	0.0000
Netflix Stream	56	0.1667	0.0000	0.0000	0.0000	**0.9200**
HTTP	124724	0.9840	**0.9962**	0.8832	0.9942	0.9869
DNS	184495	0.9931	**1.0000**	0.9996	1.0000	0.9951
SSH	102	0.0092	0.0000	0.0204	0.0000	**0.9579**
Telnet	4453	0.0031	0.0000	**0.9931**	0.0000	0.9879
NNTP	0	0.0000	0.0000	0.0000	0.0000	0.0000
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	1449	0.6504	0.9090	**0.9120**	0.8951	0.8225
Oscar File	0	0.0000	0.0000	0.0000	0.0000	0.0000
Oscar IM	3106	0.9186	0.6340	**0.9823**	0.6902	0.9559
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	694	0.0029	0.2934	**0.8812**	0.0670	0.8046
Netflix Auth	16	0.0000	0.0000	0.0000	0.0000	0.0000

Table 6.7: Comparison of the accuracy achieved by each Machine Learning Agent when trained and tested with a 1:1 and 1:3 ratio of samples for Benign Traffic Set 2. Also an indicator of how many samples for each protocol the agent was tested on.

published by the Packet Manager Agent. The latency values here are discouraging though, as the delay indicates that the system will take on average on the order of hundreds of milliseconds to respond, and at worst, even with the fastest Machine Learning Agent, 1 to 3 seconds. The impact of these results will be discussed further in 6.3.11

Latency	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	3728.4230	572.5233	746.8941	322.8444	222.0206
Stdv	4680.2973	856.9494	1400.6809	564.3380	335.4822
Min	-14.0000	-20.0000	-14.0000	-24.0000	-18.0000
Max	13980.0000	3210.0000	6112.0000	3106.0000	1821.0000

Latency	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	2217.7676	888.8405	331.7433	142.0648	308.8558
Stdv	3228.7942	1292.8086	566.7469	206.7968	429.1985
Min	-69.0000	-14.0000	-31.0000	-204.0000	-46.0000
Max	9446.0000	4557.0000	2780.0000	1243.0000	2332.0000

Latency	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	1287.4807	496.5020	176.3526	154.6487	429.1113
Stdv	1797.6638	741.0095	280.8980	257.7050	569.9073
Min	-4.0000	-19.0000	-25.0000	-82.0000	-15.0000
Max	5784.0000	2969.0000	1484.0000	1336.0000	2220.0000

Latency	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	671.8078	506.7669	167.5678	213.8990	275.1889
Stdv	975.3876	688.6729	277.5858	307.9939	409.0836
Min	-21.0000	-17.0000	-113.0000	-21.0000	-17.0000
Max	3292.0000	2566.0000	1544.0000	1462.0000	1880.0000

Table 6.8: Comparison of the latency of the system from the time the Logger Agent receives a feature to the time the Logger Agent receives an associated label from the Machine Learning Agent, when trained on various training:testing ratios for Benign Traffic Set 2. All latency values are in milliseconds.

6.3.8 Benign Traffic Set 3

Trials 41 to 60 involved the Benign Traffic Set 3 and the same Snort rules used for the previous experiment.

6.3.9 Results

The charts in Figures 6.9 to 6.12 are the most dynamic so far and demonstrate how well the Machine Learning Agents compensate over time to the changing nature of the traffic. The trend for the first 10% of the traffic is similar to the first 20% of the traffic in the previous trials. Each of the Machine Learning Agents is capable of identifying

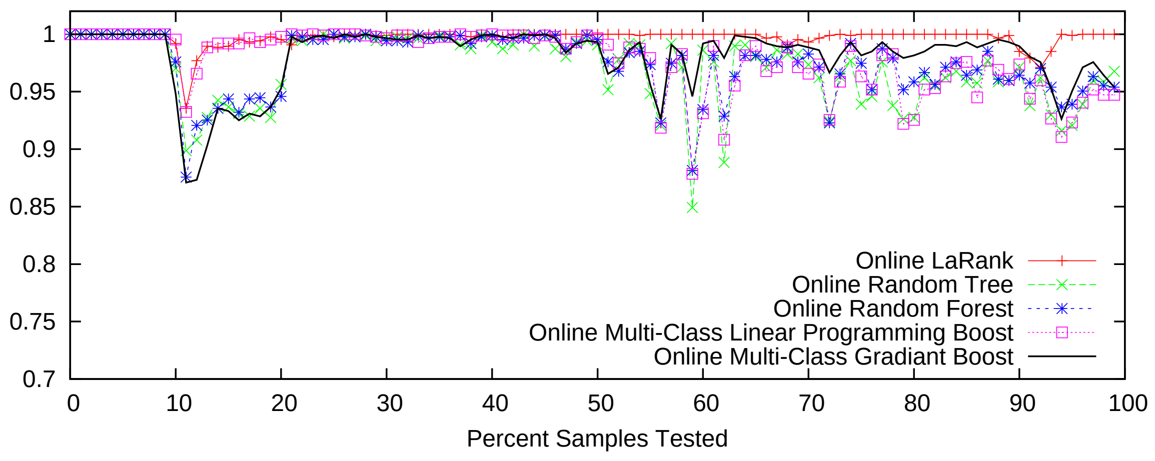


Figure 6.9: Accuracy results on Benign Traffic Set 3. Trained on 90% and Tested on 10%. The y-axis represents the test accuracy.

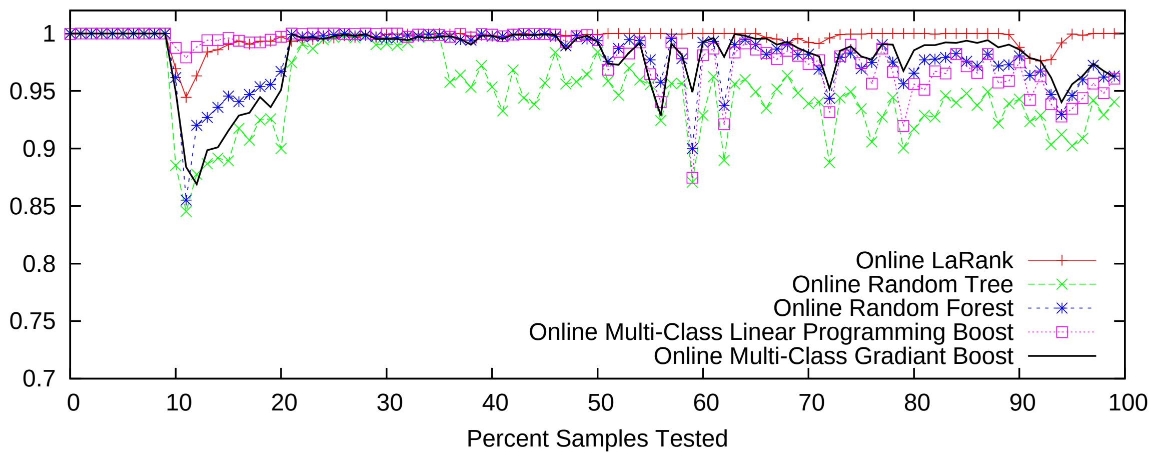


Figure 6.10: Accuracy results on Benign Traffic Set 3. Trained on 75% and Tested on 25%. The y-axis represents the test accuracy.

the DNS accurately as the clients prepare to begin other various protocol connections. As with the previous trials, the Machine Learning Agents recover somewhat after a period (as demonstrated at the 20% mark in the charts). However, after continuing to run accurately until roughly 40% of the samples have been processed, the Online Random Forest, Online Random Tree, and Online Multi-Class Linear Programming Boost show a significant degrade in performance. The Online Multi-Class Gradient

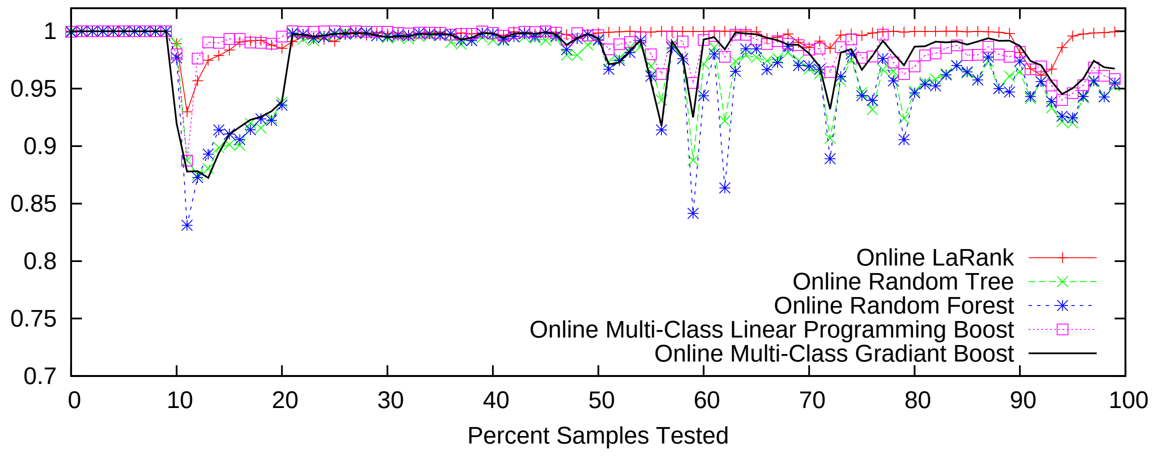


Figure 6.11: Accuracy results on Benign Traffic Set 3. Trained on 50% and Tested on 50%. The y-axis represents the test accuracy.

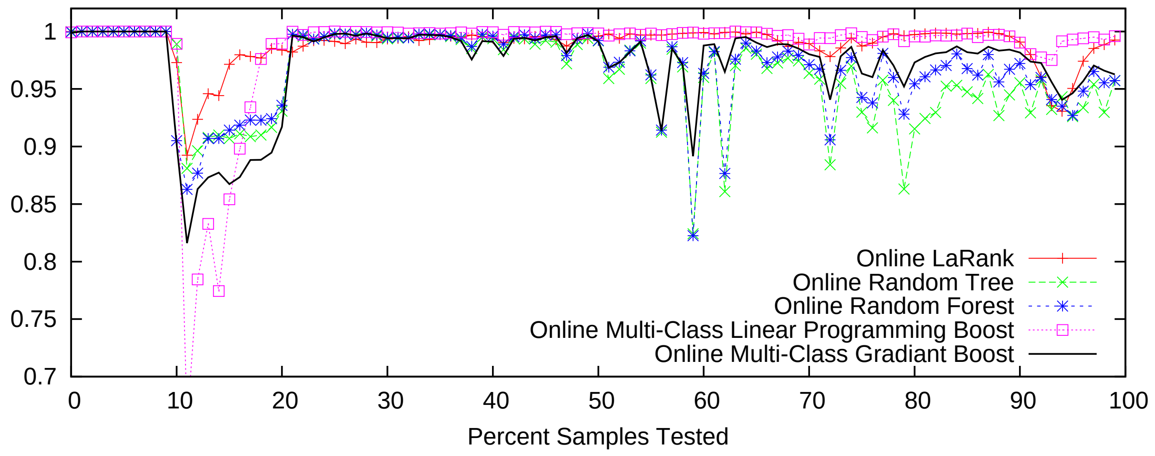


Figure 6.12: Accuracy results on Benign Traffic Set 3. Trained on 25% and Tested on 75%. The y-axis represents the test accuracy.

Boost Agent also degrades somewhat. However, the LaRank manages to maintain a relatively high level of accuracy with a couple of dips at both 70% and 90%.

The overall accuracy of each of the Machine Learning Agents is comparable to the trials on Benign Traffic Set 2 (see Table 6.9). As with previous trials, the LaRank Machine Learning Agent has a consistently higher overall accuracy as well as a smaller standard deviation. While at the 99$^{\text{th}}$, 75$^{\text{th}}$ and 50$^{\text{th}}$ percentile, the other Machine

9:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9699	0.9740	0.9773	0.9805	0.9970
Stdv	0.0088	0.0082	0.0063	0.0095	0.0021
99th pct	0.9586	0.9551	0.9472	0.9654	1.0000
75th pct	0.9770	0.9919	0.9897	0.9931	0.9988
50th pct	0.9942	0.9988	0.9989	0.9942	1.0000

3:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9493	0.9790	0.9813	0.9802	0.9961
Stdv	0.0144	0.0077	0.0060	0.0099	0.0038
99th pct	0.9293	0.9621	0.9482	0.9675	1.0000
75th pct	0.9493	0.9830	0.9903	0.9888	0.9995
50th pct	0.9644	0.9951	0.9985	0.9981	0.9990

1:1	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9676	0.9661	0.9868	0.9778	0.9944
Stdv	0.0108	0.0116	0.0044	0.0113	0.0030
99th pct	0.9431	0.9428	0.9612	0.9689	0.9988
75th pct	0.9750	0.9802	0.9928	0.9844	0.9975
50th pct	0.9881	0.9973	0.9963	0.9975	0.9965

1:3	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Accuracy	0.9620	0.9683	0.9840	0.9710	0.9884
Stdv	0.0113	0.0113	0.0179	0.0151	0.0058
99th pct	0.9297	0.9555	0.9930	0.9659	0.9885
75th pct	0.9696	0.9772	0.9985	0.9865	0.9944
50th pct	0.9940	0.9980	0.9987	0.9968	0.9937

Table 6.9: Comparison of the accuracy of the five Machine Learning Agents when trained on various training to testing ratios for Benign Traffic Set 3.

Learning Agents may have a marginally higher accuracy rate, their relatively higher standard deviations and lower overall accuracy indicate that their performance is not as reliable as the LaRank Machine Learning Agent.

Given the larger volume of traffic, and therefore higher number of instances of each individual protocol, the Table 6.10 and 6.11 are fairly good measures of the true underlying performance of the Machine Learning Agents with respect to the specific protocols involved. First, the LaRank Machine Learning Agent is capable of identifying more protocols, for example in the 1:3 ratio trials only LaRank is capable of identifying instances of BitTorrent Tracker, FaceBook, Netflix Streaming, and SSH.

	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	2	0.0000	0.0000	0.0000	0.0000	0.0000
Gmail	1076	0.9964	0.9198	**0.9981**	0.9520	0.9952
BitTorrent Track	37	0.0000	0.0000	0.0000	0.0000	**0.7812**
FaceBook	2	0.0000	0.0000	0.0000	0.0000	**0.5000**
Netflix Stream	15	0.0000	0.0000	0.0000	0.0000	**1.0000**
HTTP	35900	0.9985	0.9996	0.9980	**0.9992**	0.9974
DNS	45086	1.0000	0.9999	**1.0000**	**1.0000**	0.9984
SSH	29	0.0000	0.0000	0.0000	0.0000	**0.9375**
Telnet	2045	0.1314	0.2144	0.3619	0.4741	**0.9976**
NNTP	219	0.0000	0.0000	0.0000	0.0000	**0.9318**
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	612	0.8054	0.9480	0.9535	0.9503	**0.9642**
Oscar File	424	0.7404	0.8867	0.8916	0.8685	**0.9715**
Oscar IM	1375	0.8158	0.9527	0.8503	0.9362	**0.9883**
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	130	0.8889	0.5812	0.8522	0.5944	**0.9549**
Netflix Auth	11	0.0000	0.0000	0.1111	0.0000	**1.0000**

	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	3	0.0000	0.0000	0.0000	0.0000	**0.6000**
Gmail	2560	0.3817	0.9245	0.9858	0.9543	**0.9969**
BitTorrent Track	75	0.0000	0.0000	0.0000	0.0000	**0.8333**
FaceBook	6	0.0000	0.0000	0.0000	0.0000	**0.6000**
Netflix Stream	36	0.0000	0.0000	0.1064	0.0000	**1.0000**
HTTP	84978	0.9615	0.9992	**0.9996**	0.9989	0.9963
DNS	106789	0.9975	0.9994	0.9999	**1.0000**	0.9979
SSH	58	0.0000	0.0000	0.0159	0.0000	**0.8644**
Telnet	4873	0.3127	0.4372	0.4450	0.4966	**0.9982**
NNTP	516	0.0000	0.0000	0.0000	0.0000	**0.9082**
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	1430	0.7507	0.9474	**0.9591**	0.9449	0.9546
Oscar File	1028	0.5963	0.8185	0.9076	0.8723	**0.9510**
Oscar IM	3183	0.8876	0.9479	0.9401	0.9175	**0.9894**
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	303	0.3586	0.5605	0.8702	0.5724	**0.9444**
Netflix Auth	25	0.0000	0.0333	0.0000	0.0000	**0.9259**

Table 6.10: Comparison of the accuracy achieved by each Machine Learning Agent when trained and tested with a 9:1 and 3:1 ratio of samples for Benign Traffic Set 3. Also an indicator of how many samples for each protocol the agent was tested on.

The Online Random Tree identified almost a third of the Netflix Authentication traffic, whereas LaRank achieved a 0.9294 accuracy. The high overall accuracy in the previous Table is misleading, as there are several protocols that make up less than one percent of the total traffic. While the other Machine Learning Agents perform marginally better when the trials contain a higher training-to-testing ratio, it is clear

	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	8	0.0000	0.0000	0.0000	0.0000	**0.5000**
Gmail	4970	0.7267	0.8941	0.9887	0.8756	**0.9966**
BitTorrent Track	156	0.0000	0.0000	0.0000	0.0000	**0.7927**
FaceBook	13	0.0000	0.0000	0.0000	0.0000	**0.6364**
Netflix Stream	71	0.0000	0.0000	0.0556	0.0000	**0.9701**
HTTP	166687	0.9918	**0.9989**	0.9971	0.9981	0.9946
DNS	209885	**1.0000**	0.9996	**1.0000**	**1.0000**	0.9972
SSH	111	0.3273	0.0000	0.0259	0.0000	**0.9483**
Telnet	9555	0.2795	0.0000	0.7037	0.4508	**0.9977**
NNTP	1026	0.0000	0.0000	0.0000	0.0000	**0.8526**
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	2814	0.7591	0.9432	**0.9516**	0.9469	0.9340
Oscar File	2059	0.6602	0.8092	0.9109	0.8199	**0.9237**
Oscar IM	6225	0.8928	0.8293	**0.9837**	0.9083	0.9779
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	574	0.9035	0.5694	0.8120	0.5630	**0.8967**
Netflix Auth	54	0.1111	0.0000	0.1064	0.0000	**0.9286**

	Mean Samples	ORT Accuracy	ORF Accuracy	OMCLPBoost Accuracy	OMCGBoost Accuracy	LaRank Accuracy
Yahoo Mail	11	0.0000	0.0000	0.0000	0.0000	0.0000
Gmail	7426	0.8455	0.9550	0.9931	0.7902	**0.9910**
BitTorrent Track	230	0.0302	0.0000	0.0000	0.0000	**0.7545**
FaceBook	20	0.0000	0.0000	0.0000	0.0000	**0.6000**
Netflix Stream	102	0.0000	0.0000	0.0187	0.0000	**0.8286**
HTTP	248273	**0.9988**	0.9978	0.9708	0.9964	0.9887
DNS	312352	**1.0000**	0.9998	0.9999	**1.0000**	0.9945
SSH	170	0.0000	0.0000	0.0000	0.0000	**0.9024**
Telnet	14289	0.0214	0.0931	0.9725	0.3724	**0.9951**
NNTP	1522	0.6214	0.0000	**0.7247**	0.0000	0.5276
BitTorrent Peer	0	0.0000	0.0000	0.0000	0.0000	0.0000
eDonkey	4196	0.7109	0.9276	**0.9389**	0.9303	0.8783
Oscar File	3050	0.5100	0.7069	**0.9204**	0.7914	0.8570
Oscar IM	9345	0.6819	0.8664	**0.9864**	0.7693	0.9565
GTalk	0	0.0000	0.0000	0.0000	0.0000	0.0000
IRC	879	0.3382	0.2714	0.7670	0.0195	**0.8436**
Netflix Auth	81	0.2740	0.0000	0.0000	0.0000	**0.9294**

Table 6.11: Comparison of the accuracy achieved by each Machine Learning Agent when trained and tested with a 1:1 and 1:3 ratio of samples for Benign Traffic Set 3. Also an indicator of how many samples for each protocol the agent was tested on.

that the LaRank Machine Learning Agents consistently outperform the other Agents on the majority of the protocols in the dataset.

The latency for trials against the third dataset (see Table 6.12) are more reasonable than trials on the previous two datasets. The mean latency for the Online Multi-Class Linear Programming Boost Agent, the Online Multi-Class Gradient Boost Agent and

Latency (9:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	611.5935	111.1225	69.8101	95.6712	68.4617
Stdv	1238.8644	211.3884	115.5285	169.3938	115.8913
Min	-7.0000	-92.0000	-19.0000	-48.0000	-13.0000
Max	6012.0000	1788.0000	982.0000	1395.0000	1016.0000

Latency (3:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	128.9026	126.6612	59.5506	92.1296	78.1454
Stdv	255.5385	269.2828	110.1525	157.9010	178.9009
Min	-42.0000	-64.0000	-33.0000	-55.0000	-46.0000
Max	2183.0000	1940.0000	1182.0000	1033.0000	1640.0000

Latency (1:1)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	118.8334	119.6691	69.1219	63.8907	512.8731
Stdv	221.5795	273.6477	124.8616	110.9605	1154.5060
Min	-11.0000	-280.0000	-21.0000	-13.0000	-78.0000
Max	1853.0000	1996.0000	1101.0000	845.0000	5132.0000

Latency (1:3)	ORT	ORF	OMCLPBoost	OMCGBoost	LaRank
Mean	148.7560	82.3473	82.0179	66.5392	82.8989
Stdv	361.3350	139.2186	166.3209	114.4430	144.7549
Min	-32.0000	-34.0000	-302.0000	-47.0000	-23.0000
Max	2791.0000	1222.0000	1265.0000	986.0000	1193.0000

Table 6.12: Comparison of the latency of the system from the time the Logger Agent receives a feature to the time the Logger Agent receives an associated label from the Machine Learning Agent, when trained on various training:testing ratios for Benign Traffic Set 2. All latency values are in milliseconds.

the LaRank Agent are all below 100 milliseconds aside from the 1:1 ratio trials for LaRank, where the Mean jumps to 512 milliseconds. The maximum latency is also reasonable - closer to 1 second - with only a couple of instances of maximum values over 3 seconds. At this point, it is not clear if the latency is directly due to the Machine Learning Agent, or something subtle with the way the Logger Agent is pulling messages from the AMQP server. I will discuss further in section 6.3.10.

6.3.10 Latency Experiments

It is important to support the claim that the architecture performs tasks quickly enough to manipulate traffic in an online environment. The latency results presented up to this point are discouraging. While there were instances where the Traffic Manipulation Agent reacted in milliseconds, other runs experienced seconds of delay. Additionally, some of the trials experienced negative latency. A major requirement of the Traffic Source Agent is providing an interface for higher level programming languages to manipulate low level packet structures to enable efficient analysis and the capability to insert modified packets back into the network with low latency.

In order to determine the source of the latency I performed a series of experiments involving a Traffic Manipulation Agent, an Alert Source Agent, a Feature Source Agent, and a Machine Learning Agent. The task for the agents was relatively simply. An application makes a series of DNS requests to one of five DNS servers located in different parts of the world. The host names are randomly chosen from four lists consisting of: malicious domain names, random domain names, educational institution domain names, and whitelisted domain names. Signatures exist for a subset of those domain names identifying them as belonging to the lists above. The Machine Learning Agent uses feedback from the Alert Source Agent to classify the various domain name requests as either malicious or benign. When a malicious domain lookup is identified, the Machine Learning Agent broadcasts the identification to the other participating agents, and the Traffic Manipulation Agent, on receiving a tip that a malicious DNS request was made, attempts to respond to the request with a crafted DNS response packet before the DNS server specified in the request.

Server	Latency
DNS Server A	15 ms
DNS Server B	100 ms
DNS Server C	15 ms
DNS Server D	32 ms
DNS Server E	25 ms

Table 6.13: Initial testing to determine the average latency for DNS lookups to various DNS servers. DNS Server A belongs to the local internet service provider.

After some initial testing I determined that the average DNS request/response latency for the chosen DNS servers varied between 15 to 100 milliseconds, which appeared to be dependent on the location of the DNS server, where the local internet service provider's DNS server responded the fastest at 15 milliseconds. Table 6.13 shows the various average latency calculated over 1000 DNS requests each.

The goal of the experiments was to determine the source of latency and optimize the system so that it had a reasonable chance of manipulating DNS traffic quick enough to eliminate malicious DNS requests. In each experiment the DNS application primes the system by making 80 DNS requests that the Alert Source Agent triggers on. Then the DNS application makes 1000 DNS requests choosing host names from one of the four lists previously mentioned. Each list has a 25% chance of being chosen as the source of the host name for the next DNS request.

Initial results were similar to the previous experiment results discussed in Section 6.3. However, the simplification of the experiment demonstrated that very often the Traffic Manipulation Agent would be roughly 2000 milliseconds late with the crafted response packet. Further testing revealed that two packet buffers were impacting the capability of the system to respond in a timely manner. First, the Snort intrusion

Experiment	Packets Inserted	Average Latency	Standard Deviation
Experiment 1	328	7.7866 ms	3.1830 ms
Experiment 2	305	7.8393 ms	3.1670 ms
Experiment 3	313	8.0639 ms	3.2493 ms
Experiment 4	275	8.2036 ms	3.2275 ms
Experiment 5	311	8.4469 ms	3.1228 ms

Table 6.14: Latency results for five DNS manipulation experiments.

detection system has a buffer that captures a set amount of packets before passing them up to the detection component of the engine. This often introduced a delay of several hundred milliseconds. When very low rates of traffic were passed through the system the delay was consistently 1000 milliseconds as the traffic would eventually expire from the buffer. The second packet buffer existed on the Traffic Source Agent. A similar buffer in winpcap would also delay packets until a specified threshold was reached before passing them up to the main processing loop of the Traffic Manipulation Agent.

By modifying thresholds on those buffers, and through various other code efficiency improvements the system was capable of triggering on malicious DNS requests, and inserted crafted DNS replies in under 13 milliseconds. Table 6.14 shows the latency for five DNS manipulation experiments. In all five experiments, each time the Machine Learning Agent indicated that there was a malicious DNS request, the Traffic Manipulation Agent was capable of inserting a DNS response before any of the DNS servers, including the local internet service provider's DNS server, could reply to the DNS request.

The experiments described above demonstrate that the architecture is capable of responding to incidents in network traffic quick enough to insert modified packets back into the network.

6.3.11 Discussion

There are several issues worthy of discussion here. First, does the architecture perform the described tasks by reading network traffic, distributing features to agents and labelling traffic based on the derived features? The trials were performed on an accurate representation of network traffic. The only manipulation of the traffic was performed by the agents in the system at rates of between 20 to 50 mbps, which is a reasonable network speed. The system operated with a distributed set of agents and the features distributed to the other agents were verified against the pcap files to ensure they accurately represented the traffic. The trials do in fact validate the first claim.

Second, do the experiments support the claim that a machine learning algorithm can learn to classify traffic based on traffic labelled by misuse detection? The performance of the Machine Learning Agents demonstrates that the choice of the algorithm is important for achieving a high level of accuracy in this domain. While the literature reviewed for this thesis claim very high performance accuracy in a number of domains for the algorithms tested, it is clear that the LaRank Machine Learning agent distinguished between the protocols more reliably than the other Machine Learning Agents in a number of trial scenarios. I am confident that the trials confirm that the Machine Learning Agents in this system are suitable for classifying network traffic in an online environment for malicious behaviour discovery.

Third, does the architecture perform the tasks quickly enough to manipulate traffic in an online environment? Initially the trial results were not clear on this point. While there were certainly instances where a Traffic Manipulation Agent could react within milliseconds of a feature set being published and subsequently classified by a Machine Learning Agent, there were obviously several scenarios where a delay of seconds would be too slow to manipulate the session in question. However, after analyzing the initial results, follow up experiments (See Section 6.3.10) demonstrated there are significant opportunities for improving the reaction times of the Agents in the system and optimizing system parameters. Given that the Logger Agent was required to read every feature, label and log message sent from every agent in the system and then write them to disk, there are several places where lag could have occurred. For example consider the instances were the delay was reported by a negative number. I can only assume that at some point in the system the Logger Agent could not keep up with the incoming feature messages, and while the feature messages queue at the AMQP feature exchange piled up, it was capable of clearing the relatively fewer messages in the labelled exchange.

In summary, I am confident that the the trials performed validate the capabilities of the system.

6.4 Focused Behaviour Experiments

6.4.1 Purpose

The last experiment focused on testing the Multi-Agent Malicious Behaviour Detection system's capability to label benign traffic. The following experiments are targeted at validating the Multi-Agent Malicious Behaviour Detection implementation's ability to identify repeated behaviour by malicious software agents, and the effect on the system's capability to identify malicious behaviour when the ratio of benign-to-malicious traffic changes.

6.4.2 Methodology

One of the outcomes of the first set of experiments was the dominance of the LaRank Machine Learning Agent when classifying benign traffic. While it would be possible to use a variety of Machine Learning Agents, for the purposes of validating my framework, additional agents would require a number of additional experiment trials that are unlikely to outperform the same trial setups with the LaRank Machine Learning Agent alone. Therefore, the following experiments rely on only LaRank Machine Learning Agents. Otherwise, the same Agents were used in these experiments as previous experiments, with the addition of HTTP Feature Source Agents for processing HTTP traffic and DNS Feature Source Agents for processing DNS traffic. As new techniques are developed for Machine Learning Agents, these trials can be repeated to evaluate the new Machine Learning Agents against the LaRank Machine Learning Agent.

All the traffic samples for the following trials were derived from the *Malicious Traffic Set* (Section 6.2.3). More details for the traffic samples are described with each trial description below. The traffic samples are passed to Snort 2.9.1.2 with the Sourcefire VRT Certified Rules snapshot 2919 from February 2, 2012. The rule set contains 6128 unique snort rules. I used the default Snort configuration packaged with the Sourcefire VRT Certified Rules. The resulting unified alert file was used as the misuse detection component of the system, with the rule hits translated to classes for the Machine Learning Agent.

Traffic passes through the system of agents in a similar manner as that described in Section 6.2, with the addition of features generated by both the DNS and HTTP Agents.

6.4.3 Performance Evaluation

The primary method for evaluating the performance of the system was the accuracy of the Machine Learning Agents given by the number of malicious instances of traffic identified divided by the total number of malicious traffic samples of a specific type identified by Snort. Additionally, the number of recommendations that the system makes for suspicious traffic is an indicator of how well the system has generalized the features of the malicious behaviours. The system is designed to direct network defenders to traffic that appears suspicious given that it is similar to some malicious traffic identified by the misuse detection engine. However, if too many traffic instances are identified as similar to the true malicious traffic, the network defender will be unable to verify the suspicious traffic. Additionally, a large number of recommendations is indicative of false positives.

6.4.4 Malware Update Detection

Several SMTP traffic sets were engineered from the larger Malicious Traffic Set described earlier by extracting all port 25 traffic, which included almost exclusively

SMTP traffic. Among the traffic the misuse detection system identified eight unique instances of an attack that triggered an alert from Snort rule ID 19724. The details of the rule are below:

> alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg: "POLICY attempted download of a PDF with embedded Flash over smtp"; flow: to_server, established; content: "cmVhbQ"; fast_pattern: only; pcre: "/cmVhbQ[opqr][A-Za-z0-9_\x2f]/s"; reference: bugtraq, 35759; reference: bugtraq, 44503; reference: cve, 2009-1862; reference: cve, 2010-3654; reference: url, `blogs.adobe.com/psirt/2009/07/potential_adobe_reader_and_fla.html`; classtype:policy-violation; sid: 19274; rev:1;)

Based on the existing sessions, the engineered traffic is intended to emulate a malicious multi-agent system attempting to download an update using SMTP from a malicious server. A total of five SMTP traffic sets were derived by first extracting the malicious sessions from the original SMTP traffic sample, and then blending different amounts of the malicious network traffic sessions back into the SMTP traffic. Malicious traffic sessions were inserted at random points into the SMTP traffic sample. The details of the datasets are provided in the first four rows of Table 6.15. *Total Sessions* identifies the total number of traffic sessions in the dataset. *Identified SMTP* shows the number of sessions that the Alert Source Agent identified as SMTP. The number of labelled SMTP sessions was fixed in advance such that the Alert Source Agent would only identify some of the SMTP traffic. This was done to verify that the Machine Learning Agents could learn to distinguish benign SMTP from malicious traffic. The *Unlabelled* row indicates the number of sessions that were not labelled by

	SMTP1	SMTP2	SMTP3	SMTP4	SMTP5
Total Sessions	72079	55557	52807	50607	50112
Identified SMTP	10011	10011	10011	10011	10011
Unlabelled	60066	45046	42546	40546	40096
Misuse Identified Malicious	22022	5500	2750	550	55
ASA Identified Malicious	2002	500	250	50	5
MLA Identified Malicious	21984	6112	3549	987	436
MLA Verified	21424	5230	2538	246	10
MLA Recommend	560	882	1011	741	426
Verified/Total Malicious	0.9728	0.9509	0.9229	0.4473	0.1818
Recommend/Total Malicious	0.0254	0.1604	0.3676	1.3472	7.7454

Table 6.15: Results of experiments to identify malicious software agent updating behaviour in SMTP network traffic.

the Alert Source Agent. *Misuse Identified Malicious* indicates the number of sessions that the Snort misuse detection engine identified as malicious using the Snort rule 19274 detailed above.

The Alert Source Agents were set to add labels to only 10% of the sessions identified as malicious by the misuse detection engine. This effectively provided 10% of the malicious traffic as training data, leaving the other 90% for testing. For each dataset, the number of feature sets labelled as malicious by the Alert Source Agent is identified by the *ASA Identified Malicious* row.

6.4.5 Results

In the cases where the malicious traffic made up more than 5% of the total traffic, and the Alert Source Agent identified 0.47% of it as malicious, the Multi-Agent Malicious Behaviour Detection implementation performed fairly well. The system achieved over 90% detection accuracy. The number of recommendations remained relatively low, given the number of sessions processed. Manually verifying the traffic sessions identified as suspicious, but not verified by the misuse detection engine, would still require significant effort from the network defender. However, as fewer and fewer malicious sessions were identified, the system made more recommendations relative to the number of actual malicious instances. In dataset SMTP 5, only 10 malicious sessions were positively identified, while the system recommended 426 for further analysis. Note, however, that the 426 recommended did not overlap with the 55 instances of malicious traffic identified by the misuse detection system. For the detection of this particular updating behaviour, the Multi-Agent Malicious Behaviour Detection implementation would require that somewhere between 0.009% and 0.47% of the traffic was identified as malicious by the misuse detection system.

6.4.6 Malware Propagation

The following experiment involve network traffic containing three unique malicious attacks identified by the following Snort misuse detection rules:

> alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg: “SQL union select - possible sql injection attempt - GET parameter”; flow: established, to_server; content: “union”; fast_pattern; nocase; http_uri; content: “select”; nocase; http_uri; pcre: “/union\s + (all\s +)?select\s + [\^ /\ \] + from \s+[\^/\\]+/Ui”; metadata: policy security-ips drop, service http; classtype:misc-attack; sid: 13990; rev: 8;)

> alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS (msg: “WEB-MISC Generic HyperLink Buffer Overflow attempt”; flow: to_server,

established; content: "GET "; nocase; isdataat: 1450, relative; pcre: "/GET \x2f[^\r\n]{1450}/"; reference: bugtraq, 13045; reference: bugtraq, 14195; reference: cve,2005-0057; reference: cve,2005-0986; classtype: attempted-user; sid: 17410; rev:1;)

alert tcp $EXTERNAL_NET $FILE_DATA_PORTS -> $HOME_NET any (msg: "SPECIFIC-THREATS Microsoft Office Excel MergeCells record parsing code execution attempt"; flow: to_client, established; flowbits: isset, file.xls; file_data; content: "|E5 00 32 00 06 00 04 00 04 00 00 00 04 00 00 00 04 00 05 00 00 02 00 00 00 00 02 00 04 00 02 00 02 00|"; fast_pattern: only; reference: bugtraq, 43652; reference: cve,2010-3237; reference: url, `technet.microsoft.com/en-us/security/bulletin/MS10-080`; classtype: attempted-user; sid:20130; rev:1;)

Each of the attacks represent a malicious software agent's attempt to propagate to additional hosts and mimic common protocols (e.g. HTTP). The dataset was generated by extracting port 80 traffic from the Malicious Traffic Set. To emulate varying degrees of malicious behaviour the malware attacks were randomly dispersed throughout the set of benign HTTP traffic to form five HTTP datasets. The first three rows of Table 6.16 provide details of the traffic samples. The structure of the table is similar to Table 6.15, except that the last seven rows are repeated for each emulated malicious software agent propagation type, signified by the Snort Signature ID of the associated Snort rule. Roughly 20% of the original HTTP traffic was labelled as HTTP by the Alert Source Agent.

The goal of the trial was to identify how much malicious HTTP traffic was required

	HTTP1	HTTP2	HTTP3	HTTP4	HTTP5
Total Sessions	273085	156562	137125	121582	118084
Identified HTTP	23544	23544	23544	23544	23544
Unlabelled	235417	129487	111817	97687	94507
Snort Signature 20130					
Misuse Identified	51788	12947	6468	1287	121
ASA Identified	4708	1177	588	117	11
MLA Identified	51819	13381	6943	2230	836
MLA Verified	51343	12788	6525	1214	97
MLA Recommend	476	593	590	1016	739
Verified/Total Malicious	0.9914	0.9877	0.9822	0.9433	0.8016
Recommend/Total Malicious	0.0091	0.0458	0.0912	0.7894	6.1074
Snort Signature 17410					
Misuse Identified	51788	12947	6468	1287	121
ASA Identified	4708	1177	588	117	11
MLA Identified	51655	12994	6673	1812	224
MLA Verified	51257	12581	6157	1163	95
MLA Recommend	398	413	516	649	129
Verified/Total Malicious	0.9897	0.9717	0.9519	0.9036	0.7851
Recommend/Total Malicious	0.0077	0.0319	0.0798	0.5043	1.0661
Snort Signature 13990					
Misuse Identified	51788	12947	6468	1287	121
ASA Identified	4708	1177	588	117	11
MLA Identified	52138	13647	7155	2077	623
MLA Verified	51163	12586	6194	1087	30
MLA Recommend	975	1061	961	990	593
Verified/Total Malicious	0.9879	0.9721	0.9576	0.8446	0.2479
Recommend/Total Malicious	0.0188	0.0819	0.1486	0.7692	4.9001

Table 6.16: Results of experiments to identify malicious software agent propagating over HTTP.

for the Multi-Agent Malicious Behaviour Detection implementation to distinguish between benign HTTP and malicious HTTP propagation. The feature set for HTTP provides additional features over the TCP feature sets used in the previous experiments (118 to 25 respectively), providing additional features to Machine Learning

Agents, which should in turn increase classification accuracy. Table 6.16 shows the results of these trials.

6.4.7 Results

The Multi-Agent Malicious Behaviour Detection implementation maintained a high classification accuracy in the first four HTTP datasets for each of the three malicious HTTP propagation traffic types, maintaining over 90% accuracy with the exception of signature ID 13990 on HTTP dataset 4. Even the fifth dataset, where the labelled malicious samples were 0.009% of the total traffic (11/118084), the Multi-Agent Malicious Behaviour Detection implementation achieved 80% and 78% accuracy for signature ID 20130 and 17410. The recommendations stayed fairly steady across all of the trials, indicating that some traffic consistently shared similar features to the malicious traffic. This is to be expected, since HTTP is commonly exploited, and used for a variety of applications. The 17410 traffic resulted in a low number of recommendations, which signifies a high likelihood that the recommendations actually indicate a variation on the original malicious traffic, or benign traffic that is very similar to the malicious behaviour.

6.4.8 Malware over UDP

This experiment is intended to demonstrate the Multi-Agent Malicious Behaviour Detection's performance in identifying propagation behaviour from malicious software agents over UDP. The datasets were derived from all of the UDP traffic contained in the Malicious Traffic Set, amounting to almost 500 megabytes of traffic. The malicious traffic chosen to represent propagation over UDP were identified by the following Snort signatures:

> alert udp any any -> any 69 (msg: “TFTP GET filename overflow attempt”; flow: to_server; content: “|00 01|”; depth: 2; isdataat: 100, relative; content: ¡‘|00|”; within: 100; metadata: policy balanced-ips drop, policy security-ips drop, service tftp; reference: bugtraq,22923; reference: bugtraq, 36121; reference: bugtraq, 5328; reference: cve, 2002-0813; reference: cve, 2009-2957; reference: nessus, 18264; classtype: attempted-admin; sid: 1941; rev: 14;)

> alert udp any 4000 -> any any (msg: “EXPLOIT ICQ SRV MULTI/SRV META USER overflow attempt”; flow: to_server; content: “|05 00|”; depth:2; content: “|12 02|”; within: 2; distance: 5; byte_test: 1, >, 1 , 12, relative; content: “|05 00|”; content: “n|00|”; within: 2; distance: 5; content: “|05 00|”; content: “|DE 03|”; within: 2; distance: 5; byte_test: 2,>, 512, -11, relative, little; reference: cve,2004-0362; reference: url, `www.eeye.com/html/Research/Advisories/AD20040318.html`; classtype: misc-attack; sid: 2446; rev: 11;)

Both signatures represent overflow attempts, that could provide the attacking malicious software agents with elevated privileges to facilitate propagation. Given the large number of traffic sessions, blending in the malicious traffic was not as exact as in the previous experiments. The unique malicious sessions identified were extracted from the original network traffic and blended into a set of just UDP traffic with a

	UDP1	UDP2	UDP3	UDP4	UDP5
Total Sessions	1937729	1424022	1318629	1214085	1190567
Identified UDP	237594	237594	237594	237594	237594
Unlabelled	1631789	1165011	1069157	974117	952737
Snort Signature 1941					
Misuse Identified	249919	78684	65329	13057	1298
ASA Identified	22782	7139	5939	1187	118
MLA Identified	690754	477888	463605	335570	254054
MLA Verified	246878	76485	63106	11725	712
MLA Recommend	443876	401403	400499	323845	253342
Verified/Total Malicious	0.9878	0.9721	0.9660	0.8980	0.5485
Recommend/Total Malicious	1.7760	5.1015	6.1305	24.8024	195.1787
Snort Signature 2446					
Misuse Identified	499839	157367	65329	13057	1298
ASA Identified	45564	14278	5939	1187	118
MLA Identified	496472	154972	64193	12660	1279
MLA Verified	496267	154799	63980	12439	1170
MLA Recommend	205	173	213	221	109
Verified/Total Malicious	0.9929	0.9837	0.9794	0.9527	0.9014
Recommend/Total Malicious	0.0004	0.0011	0.0033	0.0169	0.0840

Table 6.17: Results of experiments to identify malicious software agents propagating over UDP.

goal of achieving roughly 4:10, 2:10, 1:10, 2:100, and 1:1000 malicious-to-benign traffic ratios. However, the traffic was not split evenly, resulting in more malicious TFTP traffic than ICQ traffic. The Alert Source Agent was set to identify roughly 10% of the malicious traffic with a label associating it with either Snort signature 1941 or 2446. Table 6.17 shows the details of the traffic and has the same general format as Table 6.16.

6.4.9 Results

The Multi-Agent Malicious Behaviour Detection implementation was able to detect the malicious TFTP traffic consistently, with the accuracy decreasing slightly until UDP dataset five, where the accuracy drops from 0.8980 to 0.5485. However, the Multi-Agent Malicious Behaviour Detection implementation makes an overwhelming number of recommendations for suspicious traffic. In the first dataset, the Multi-Agent Malicious Behaviour Detection implementation flags almost twice as many sessions as the misuse detection system, and by the fifth UDP dataset it flags almost 200 times more sessions than the misuse detection system. The number of false positives would surely overload a network defender, significantly lowering the value of the system's capability to identify the truly malicious traffic. For UDP dataset 1, the network defender would have to determine which of the 690754 classified samples were actually malicious (246878) and which were likely false positives (443876). The multitude of false positives here is likely due to the similarity of the traffic across the chosen feature set. The false positives are an indication that the feature set does not contain enough features for the machine learning algorithm to distinguish between the malicious and the benign traffic. As discussed in Section 3.7, there are a variety of potential features to choose from for identifying malicious communications. While the subset of features I have chosen were identified in the literature as sufficient for tracking malicious communications, they were not sufficient for the traffic here. I will discuss this further in Chapter 7.

The Multi-Agent Malicious Behaviour Detection implementation identifies the malicious ICQ traffic with a high accuracy across all five datasets, achieving over

90%. Also, unlike many previous experiments, the number of recommendations stays at a relatively manageable level. The ratio between recommended session and verified malicious sessions stays low, indicating that either there is another protocol in the traffic that has very similar features to the malicious ICQ traffic, or there is a second variation of the malicious ICQ traffic in the dataset.

6.4.10 Exfiltrate, Beacon, and Update

The following experiment involved the extraction of all TCP traffic from the Malicious Traffic Set (1.805 gigabytes), and the blending of three unique malicious behaviours into the resulting set to simulate communications between agents in a malicious multi-agent system. The malicious behaviours were chosen to represent beaconing over IRC, exfiltrating as in a P2P protocol, and updating by downloading an infected PDF document (Section 1.5.4). Effectively, the actual malicious behaviours that exist in the Malicious Traffic Set are multiplied to simulate the collaboration between malicious software agents. Instead of a few instances of each malicious behaviour, as in the original Malicious Traffic Set, the derived set contains several thousand instances of each. As malicious software agents attempt to communicate to their peers or superiors, the Multi-Agent Malicious Behaviour Detectionsystem learns to identify more and more instances. The following signatures identified the malicious traffic in the original dataset:

> alert tcp $EXTERNAL_NET any -> $SMTP_SERVERS 25 (msg: “POLICY attempted download of a PDF with embedded Flash over smtp”; flow: to_server, established; content: “cmVhbQ”; fast_pattern: only;

pcre: "/cmVhbQ[opqr][A-Za-z0-9_\x2f]/s"; reference: bugtraq, 35759; reference: bugtraq, 44503; reference: cve, 2009-1862; reference: cve, 2010-3654; reference: url, `blogs.adobe.com/psirt/2009/07/potential_adobe_reader_and_fla.html`; classtype: policy-violation; sid: 19274; rev: 1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET 6666:7000 (msg: "CHAT IRC channel join"; flow: to_server, established; content: "JOIN "; fast_pattern: only; pcre: "/^\s*JOIN/smi"; metadata: policy security-ips drop; classtype: policy-violation; sid: 1729; rev: 8;)

alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg: "P2P BitTorrent transfer"; flow: to_server, established; content: "|13|BitTorrent protocol"; depth: 20; metadata: policy security-ips drop; classtype: policy-violation; sid: 2181; rev: 4;)

The exact number of sessions for each type of malicious traffic are provided in Table 6.18. The traffic blending evenly distributed the malicious traffic into TCP datasets 3, 4, and 5. However, in datasets 1 and 2, the blending technique failed to blend in the malicious PDF downloading traffic. While this particular distribution was not intended, it does provide an indication of the performance of the implementation when there are very few malicious malware samples in a large dataset. I direct the reader back to Section 6.4.4 for an example of detecting the update behaviour on larger datasets. When labelling the traffic, the sessions were labelled as TCP, signature 2818, signature 1729, signature 19274, or not labelled at all. Given the volume of TCP traffic, and the number of protocols present in the traffic, the label

	TCP1	TCP2	TCP3	TCP4	TCP5
Total Sessions	6474197	4518359	4096511	3632366	3527921
Identified TCP	703267	703267	703267	703267	703267
Unlabelled	5502110	3724126	3340501	2918551	2823601
Snort Signature 2181					
Misuse Identified	1548918	538718	193391	38676	3861
ASA Identified	140388	55739	17581	3516	351
MLA Identified	1571642	721868	389044	170660	62180
MLA Verified	692431	198380	41703	5795	224
MLA Recommend	879211	523488	347341	164865	61956
Verified/Total Malicious	0.4470	0.3682	0.2156	0.1498	0.0580
Recommend/Total Malicious	0.5676	0.9717	1.7961	4.2627	16.0466
Snort Signature 1729					
Misuse Identified	1408595	463191	193391	38676	3861
ASA Identified	128400	35219	17581	3516	351
MLA Identified	1303506	512294	449934	188974	37095
MLA Verified	537936	77783	47085	5449	116
MLA Recommend	765570	434511	402849	183525	36979
Verified/Total Malicious	0.3819	0.1679	0.2435	0.1409	0.0300
Recommend/Total Malicious	0.5435	0.9381	2.0831	4.7452	9.5776
Snort Signature 19274					
Misuse Identified	346	112	193391	38676	3861
ASA Identified	32	8	17581	3516	351
MLA Identified	4803	1617	182884	38215	4987
MLA Verified	84	4	178947	36873	3560
MLA Recommend	4803	1613	3937	1342	1427
Verified/Total Malicious	0.2428	0.0357	0.9253	0.9534	0.9220
Recommend/Total Malicious	13.8815	14.4018	0.0204	0.0347	0.3700

Table 6.18: Results of experiments to identify malicious software agent propagating over TCP behaviour.

of TCP traffic was applied to various distinct protocols. Effectively, the TCP traffic label acted as a *catchall* type of classification. The results highlight the importance of choosing a representative list of features capable of distinguishing one type of traffic from another. I will discuss the impact of the chosen features more in Chapter 7.

6.4.11 Results

The Multi-Agent Malicious Behaviour Detection implementation was unable to consistently identify the malicious traffic from signatures 2181 and 1729, achieving both low accuracy rates and high numbers of recommendations. The results indicate that several TCP sessions share similar features, such that the Machine Learning Agents could not distinguish between them well. Contrast to signature 19274 in datasets 3, 4, and 5, where the Machine Learning Agents learned to make accurate identifications of the traffic and maintain a fairly low recommendation amount. The malicious PDF traffic likely contains unique values for some of the features extracted, enabling more accurate classification.

6.4.12 Discussion

The focused behaviour experiments demonstrated the Multi-Agent Malicious Behaviour Detection implementations ability to detect some of the malicious traffic when enough samples of the traffic was available, and the features were sufficient for the Machine Learning Agents to learn to distinguish between the benign and malicious traffic. By providing an HTTP Feature Source Agent, that is capable for deriving additional features from HTTP specific traffic, the Machine Learning Agents could achieve high levels of accuracy with a low level of recommendations, making it possible for a network defender to verify the recommendations from the system. The small features sets, such as UDP and TCP, demonstrated poorer classification ability. However, the SMTP propagation experiment shows that there is benefit to splitting the traffic early on in the classification heuristically, for example by port number, to group together similar protocols. In all cases, it was clear that malicious behaviour frequency must attain a specific repetition threshold in order for Machine Learning Agents to have any chance to learn to classify them accurately. This is where artificially inserting malicious traffic back into the system via additional Traffic Source Agents for training purposes (Section 5.5.1) shows promise for improving the Multi-Agent Malicious Behaviour Detection's performance.

6.5 Complete Higher Education Network

6.5.1 Purpose

This experiment was intended to test the system's capability of identifying malicious multi-agent system attacks without limiting the amount of malicious traffic or the types of malicious traffic present. The primary difference from the first set of validation experiments and this set of experiments is the presence of malware in the traffic that is potentially unrecognizable to the misuse detection component of the system. These experiments represent a more difficult scenario, where the machine learning component works with the the other agents in the system to identify threats to the network. To be more precise, this shows how the machine learning component effects the detection capability of the system in a richer and more challenging environment.

6.5.2 Methodology

The methodology for this experiment is very similar to the previous experiment. The agents deployed were the same as the previous experiment. The data sample involved the entire Malicious Traffic Set, unmodified. In all previous experiments, the system attempted to identify benign traffic as well as malicious traffic. However, in this experiment, traffic was only labelled as malicious. The misuse detection engine feeds in malicious alerts, and does not attempt to classify benign traffic. The experiment tests the implementation's ability to classify the traffic using confidence values associated with each classification, to direct the network defender to the traffic that is most likely an example of malicious behaviour. So, while every traffic session will ultimately be classified as malicious, the confidence value will allow network defenders to prioritize their attention. That is, they can focus their efforts on cases where confidence is high, and reduce or eliminate consideration of low-confidence cases. In principle, this should reduce the amount of traffic the network defender must investigate in order to eliminate the false positives.

6.5.3 Performance Evaluation

The metric used to measure the results of this experiment was the number of traffic sessions the Multi-Agent Malicious Behaviour Detection implementation classified as suspicious and recommended for further analysis. This metric was supplemented with manual verification of a subset of the suspicious traffic identified by the system. The manual verification provides an indication of the success of real-world use of this system (i.e. what it would be like with legitimate human intervention by a trained

ASA	MLA	SigID	Signature
29417	361844	579	RPC portmap mountd request UDP
8003	629485	1952	RPC mountd UDP mount request

Table 6.19: Summary of Alert Source Agent (ASA) labels and Machine Learning Agent (MLA) classifications for UDP traffic (excluding DNS). The table only contains data on snort signatures that fired more than 5 times on the dataset.

professional).

6.5.4 Full Data Set

In this experiment the dataset consisted of the entire 3418.12 megabytes of network traffic generated for the Malicious Traffic Set. The traffic had a variety of malware attacks woven into it, as described in Section 6.2.3. In total the misuse detection engine identified 395 unique misuse detection alerts types.

6.5.5 Results

The UDP Transport Layer Feature Source Agent processed 487.39 megabytes of UDP traffic and produced 1225410 feature sets. Of those feature sets, the Alert Source Agent labelled 37439 with hits from the external misuse detection system. The misuse detection system identified two attack types (see Table 6.19). A LaRank Machine Learning Agent, subscribing to all UDP features, processed the 1225410 UDP features sets and classified each based on the 37439 alert labels, producing 1225410 labelled feature sets.

Figure 6.13 illustrates the distribution of confidence values the Machine Learning Agent assigned to each labelled feature set. The Machine Learning Agent assigned

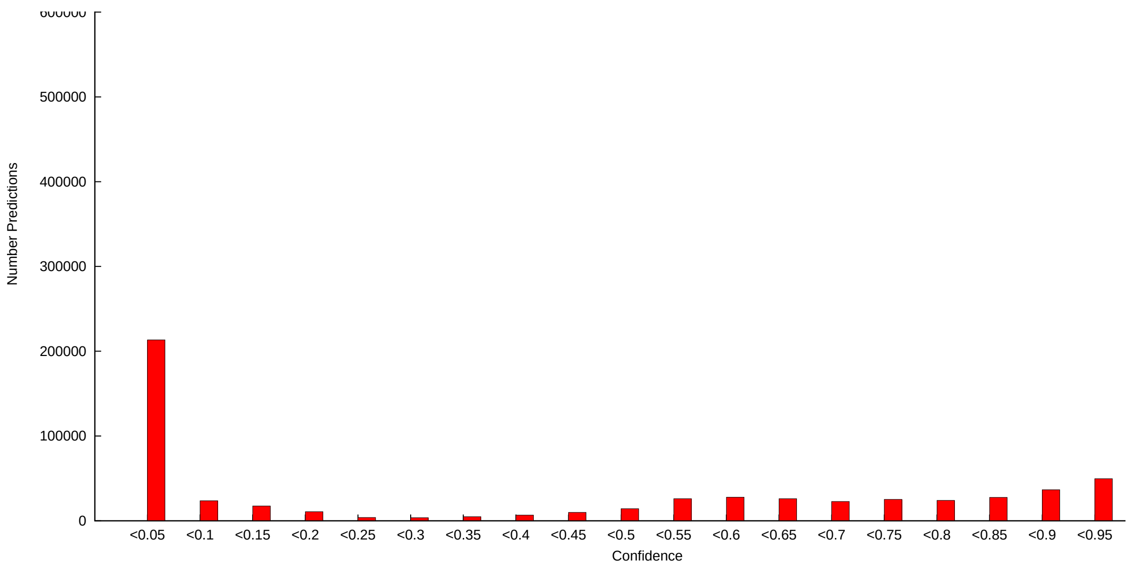


Figure 6.13: Distribution of confidence values Machine Learning Agents assigned to labelled feature sets derived from UDP traffic (excluding DNS traffic).

low confidence values to the majority of its classifications. This is an indicator to the network defender that the results are not reliable, and likely consist of many false positives. The low confidence values aid in addressing a weakness encountered earlier, that of overwhelming the network defender with false positives (Section 6.4). This provides the network defender with a mechanism to discard a large portion of traffic alerts that are likely false positives.

Table 6.20 illustrates the total number of times the Machine Learning Agent associated a feature set with a misuse detection rule. The misuse detection engine identified far fewer actual attacks, indicating either that the Machine Learning Agents are over-generalizing or are finding many attacks that the misuse detection engine has not. Table 6.19 shows the alerts that fired more than five times for the UDP traffic, with the signature ID, the count, and the full alert message. Table B.1 contains a summary of the signature IDs and counts for other rules that fired less than

Total	SigID	Rule Name
342972	1952	RPC mountd UDP mount request
182359	579	RPC portmap mountd request UDP
30989	1941	TFTP GET filename overflow attempt
28181	2088	RPC ypupdated arbitrary command attempt UDP
27514	2256	RPC sadmind query with root credentials attempt UDP
1630	15302	DOS Microsoft Exchange System Attendant denial of service attempt
1	2446	EXPLOIT ICQ SRV_MULTI/SRV_META_USER overflow attempt - ISS Witty Worm
1	648	SHELLCODE x86 NOOP

Table 6.20: Total number of feature sets associated with a specific rule by the LaRank Machine Learning Agent.

five times. Those alerts that fired less than five times were unlikely to be learned by the LaRank Machine Learning Agent, therefore I focused on those signature IDs with at least five alerts. For all but signature IDs 1952 and 579, there were less than five identified attacks. After manually parsing the traffic results, it was obvious the Machine Learning Agent was producing many false positives. The performance of the system on UDP traffic is discouraging, and reinforces the concept that in order to learn to identify malicious behaviour, the system required sufficient training samples. Given the low incidence of actual attacks, those training samples were not available.

The DNS Agent processed 305.89 megabytes of UDP traffic on port 53 and produced 2965999 feature sets. Of those features, the Alert Source Agent labelled 1483354 with hits from the external Snort misuse detection system. A Machine Learning Agent processed the 2965999 DNS feature sets and classified each one based on the 1483354 misuse detection hits, producing a total of 2965999 labelled feature sets. Table 6.21 shows that the Alert Source Agent received alerts on almost every single DNS response from the misuse detection system. When confronted with a

ASA	MLA	SigID	Signature
1482644	1803468	254	DNS SPOOF query response with TTL of 1 min. and no authority

Table 6.21: Summary of Alert Source Agent (ASA) labels and Machine Learning Agent (MLA) classifications for DNS traffic. The table only contains data on snort signatures that fired more than five times on the dataset.

situation such as this, a network defender will likely think one of three things: the network is under attack from a distributed denial of service attack by a malicious multi-agent system, the misuse detection engine is generating a significant number of false positives (in turn teaching the Machine Learning Agent to false positive on the same traffic), or the misuse detection engine is correct and the Machine Learning Agent is overgeneralizing.

The first hypothesis is plausible. Should a malicious multi-agent system attempt a distributed denial of service attack (Section 1.5.4), the number of invalid requests should be significantly higher than the number of valid requests that the network's DNS servers can manage. If the invalid requests are less than the typical number of valid requests, the impact on the network is low, and the desired effect of denying service is not achieved. In that case, the Multi-Agent Malicious Behaviour Detection system should produce far more maliciously labelled feature sets than non-malicious feature sets.

The other two hypotheses can be verified by performing checks on some of the DNS traffic to see if it is malicious, and by checking the confidence values the Multi-Agent Malicious Behaviour Detection system has applied to the feature sets. Figure 6.14 shows that, while the Machine Learning Agent classified the majority of the traffic as

DNS spoofing, the confidence for every classification remained low. The DNS packets in the Malicious Data Set generated by the Breaking Point contain features that the external misuse detection system identifies as a spoofing attack, and in turn the Machine Learning Agent learns that much of the DNS requests/responses are spoofs, given that the spoof class is the only available class provided by the misuse detection engine. The low confidence value aids in identifying issues with the underlying traffic. The DNS results demonstrate the system's reliance on an accurate misuse detection system.

This is an interesting result, since while the classifications were based on a feature specific to the traffic generation process, it has shown that the Multi-Agent Malicious Behaviour Detection would in fact flag a malicious multi-agent system denial of service attack should the system be flooded by DNS packets that were different from standard DNS. The impact of distributed denial of service on machine learning for network detection is worth further study, and I will discuss this further in Chapter 7.

The HTTP agent processed 819.70 megabytes of port 80 TCP traffic and produced 118129 feature sets. Of those features, the Alert Source Agent labelled 408 with hits from the external misuse detection system. A Machine Learning Agent processed the 118129 HTTP feature sets, and classified each one based on the 408 misuse detection hits, producing 118129 labelled feature sets. Table 6.22 shows the alerts where the Alert Source Agent matched more than five misuse detection alerts to feature sets and the resulting Machine Learning Agent classifications.

Figure 6.15 shows the distribution of confidence values for the various classifications. There are many low confidence classifications and relatively few high confidence

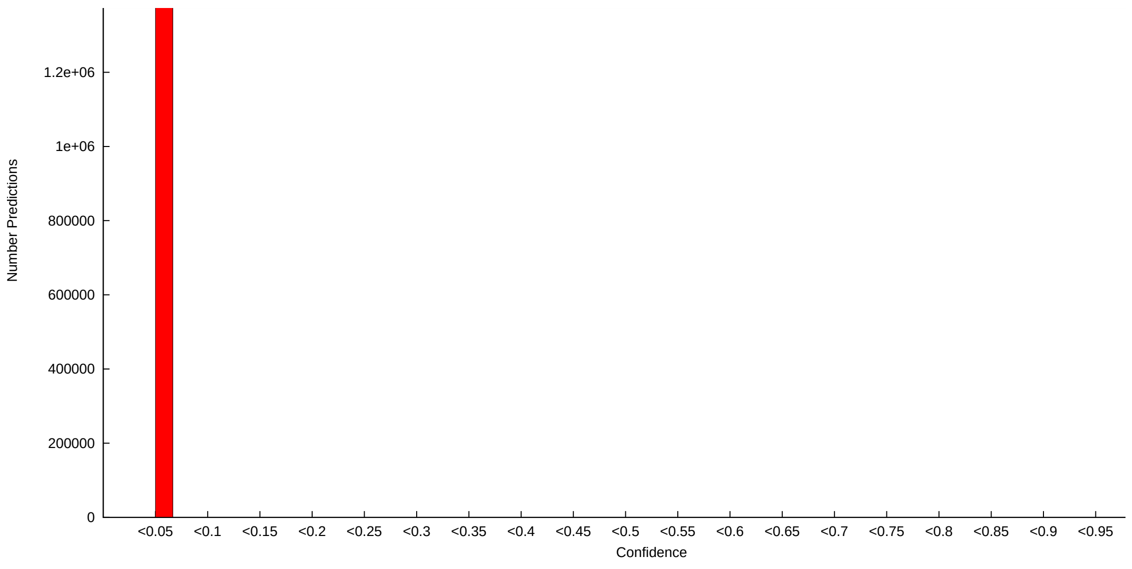


Figure 6.14: Distribution of confidence values Machine Learning Agents assigned to labelled feature sets derived from DNS traffic.

ASA	MLA	SigID	Signature
47	804	10504	SHELLCODE unescape encoded shellcode
38	91	13990	SQL union select - possible sql injection attempt - GET parameter
17	322	1394	SHELLCODE x86 inc ecx NOOP
16	83	648	SHELLCODE x86 NOOP
15	202	17410	WEB-MISC Generic HyperLink buffer overflow attempt
12	365	20130	SPECIFIC-THREATS Microsoft Office Excel MergeCells record parsing code execution attempt
12	922	7896	WEB-ACTIVEX AOL.PicEditCtrl ActiveX clsid access
12	93789	17322	SHELLCODE x86 OS agnostic fnstenv geteip dword xor decoder
10	318	10214	WEB-ACTIVEX Shockwave ActiveX Control ActiveX clsid access
9	361	1002	WEB-IIS cmd.exe access
8	34	19074	WEB-CLIENT javascript uuencoded noop sled attempt

Table 6.22: Summary of Alert Source Agent (ASA) labels and Machine Learning Agent (MLA) classifications for HTTP traffic. The table only contains data on Snort signatures that fired more than 5 times on the dataset.

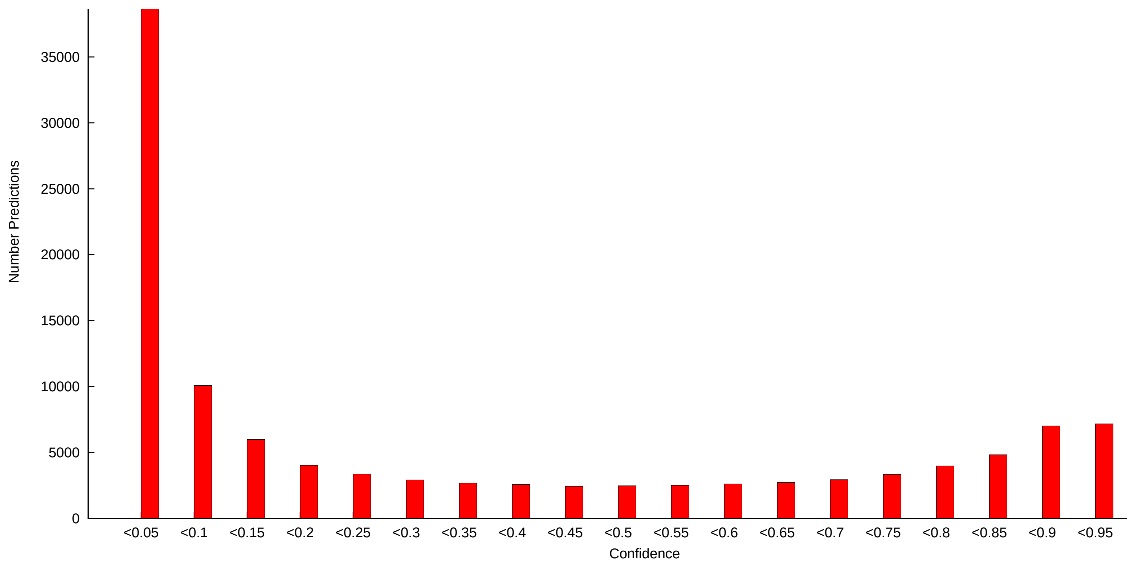


Figure 6.15: Distribution of confidence values Machine Learning Agents assigned to labelled feature sets derived from HTTP traffic.

classifications. Table 6.23 shows the alerts that were classified with a confidence of 0.90 or more. The results for the HTTP traffic, as with UDP, are plagued with false positives. Manual inspection of the sessions identified by the Machine Learning Agents showed that the Machine Learning Agent often classified benign web requests as Web-ActiveX attacks. There were some instances where the Machine Learning Agent identified what appeared to be a form of attack, however the overwhelming number of false positives outweighs the few successful classifications. I will discuss the issue of false positives more in Section 6.5.6.

The TCP Transport Layer agent processed 1805.14 megabytes of TCP network traffic and produced 3584212 feature sets. Of those features, the Alert Source Agent labelled 67874 with hits from the external misuse detection system. A Machine Learning Agent processed the 3584212 TCP feature sets and classified each based on the 67874 hits, producing 3584212 labelled feature sets. Table 6.24 shows the instances

Total	SigID	Rule Name
6989	17322	SHELLCODE x86 OS agnostic fnstenv geteip dword xor decoder
335	7502	WEB-ACTIVEX tsuserex.ADsTSUserEx.1 ActiveX clsid access
205	20650	WEB-PHP MyNewsGroups remote file include in layersmenu.inc.php myng_root
121	7896	WEB-ACTIVEX AOL.PicEditCtrl ActiveX clsid access
96	15098	WEB-ACTIVEX Microsoft Visual Basic FlexGrid ActiveX function call access
49	8066	WEB-ACTIVEX Windows Scripting Host Shell ActiveX clsid access
44	13830	WEB-ACTIVEX sapi.dll alternate killbit ActiveX clsid access
41	10214	WEB-ACTIVEX Shockwave ActiveX Control ActiveX clsid access
34	15090	WEB-ACTIVEX Microsoft Visual Basic Charts ActiveX function call access
17	8375	WEB-ACTIVEX QuickTime Object ActiveX clsid access
17	20728	WEB-PHP WoW Roster remote file include with hslist.php and conf.php
9	20130	SPECIFIC-THREATS Microsoft Office Excel MergeCells record parsing code execution attempt
8	16591	SPECIFIC-THREATS EasyMail Objects ActiveX exploit attempt - 2
5	20731	WEB-PHP TSEP remote file include in colorswitch.php tsep_config
3	1394	SHELLCODE x86 inc ecx NOOP
2	12280	WEB-CLIENT Microsoft Internet Explorer VML source file memory corruption attempt
1	7934	WEB-ACTIVEX ftp Asychronous Pluggable Protocol Handler ActiveX clsid access
1	18178	SPECIFIC-THREATS Mozilla browsers memory corruption simultaneous XPCOM events code execution attempt
1	10504	SHELLCODE unescape encoded shellcode

Table 6.23: Predictions where the Machine Learning Agent's confidence is greater than 0.90 on HTTP traffic.

were the Alert Source Agent added alert labels to more than 5 feature sets, and the resulting frequency of which the Machine Learning Agent then classified sessions as

ASA	MLA	SigID	Signature
35407	1067073	2181	P2P BitTorrent transfer
32100	872697	1729	CHAT IRC channel join
208	243151	1394	SHELLCODE x86 inc ecx NOOP
17	8483	648	SHELLCODE x86 NOOP
10	26033	12802	SHELLCODE base64 x86 NOOP
8	11398	19274	POLICY attempted download of a PDF with embedded Flash over smtp
6	1999	14737	NETBIOS DCERPC NCACN-IP-TCP host-integration bind attempt

Table 6.24: Summary of Alert Source Agent (ASA) labels and Machine Learning Agent (MLA) classifications for TCP traffic (excluding HTTP). The table only contains data on snort signatures that fired more than 5 times on the dataset.

similar to the sessions that triggered the alerts. As with other protocols, the Machine Learning Agent tended toward many false positives. However, as Figure 6.16 illustrates, it often classified the feature sets with a very low confidence value. This is as expected, as the number and variety of features used to classify the traffic is likely insufficient to distinguish between the large variety of traffic that uses TCP as a transport mechanism. As such, the Machine Learning Agent will train on a variety of very similar feature sets that the misuse detection can distinguish between using string matching.

Table 6.25 illustrates the variety of attacks over TCP, and and protocols available for attacks. Attacks varied, and included attacks against IRC Chat, SMTP, Oracle, Microsoft file transfers, and more.

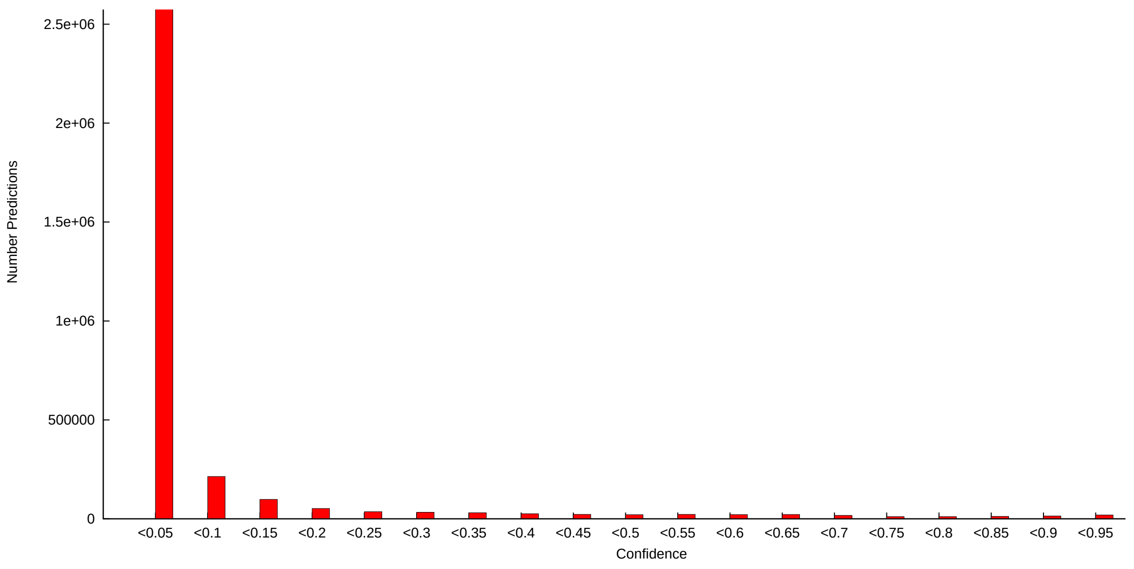


Figure 6.16: Distribution of confidence values Machine Learning Agents assigned to feature sets derived from TCP traffic (excluding HTTP traffic).

6.5.6 Discussion

Given that, in the experiment above, Machine Learning Agents could only classify traffic using labels they have previously seen, and only malicious traffic was being labelled, the Machine Learning Agents viewed all traffic as malicious and attempted to match each feature set to the closest similar malicious traffic. This resulted in an overwhelming number of false positives from the Machine Learning Agents. Each labelled feature set was also assigned a confidence value. The confidence value is intended to show that, while the Machine Learning Agent would label traffic as malicious, a higher confidence value should indicate a higher likelihood that the traffic matched the malicious traffic and was not just a false positive. While the Machine Learning Agents produced many false positives, it also clearly indicated to network defenders that it was not confident in its classifications. The confidence value helps to guide the network defender to what is likely more important traffic to analyze. As

Total	SigID	Rule Name
30596	1729	CHAT IRC channel join
19602	2181	P2P BitTorrent transfer
1206	20602	RSERVICES rlogin guest
1071	16524	FTP ProFTPD username sql injection attempt
937	604	RSERVICES rsh froot
543	18682	POLICY download of a PDF with OpenAction object
416	19280	POLICY attempted download of a PDF with embedded Flash over pop3
323	606	RSERVICES rlogin root
246	20601	RSERVICES rlogin nobody
234	2101	NETBIOS SMB Trans Max Param/Count DOS attempt
143	648	SHELLCODE x86 NOOP
121	19291	NETBIOS Microsoft LNK shortcut download attempt
86	17410	WEB-MISC Generic HyperLink buffer overflow attempt
80	20130	SPECIFIC-THREATS Microsoft Office Excel MergeCells record parsing code execution attempt
68	14737	NETBIOS DCERPC NCACN-IP-TCP host-integration bind attempt
65	16543	WEB-CLIENT Microsoft Windows Media Player codec code execution attempt
39	6584	NETBIOS DCERPC NCACN-IP-TCP rras RasRpcSubmitRequest overflow attempt
33	20662	SPECIFIC-THREATS Dameware Mini Remote Control username buffer overflow
29	19551	POLICY self-signed SSL certificate with default Internet Widgits Pty Ltd organization name
26	1394	SHELLCODE x86 inc ecx NOOP
20	19274	POLICY attempted download of a PDF with embedded Flash over smtp
13	12977	NETBIOS DCERPC NCACN-IP-TCP mqqm QMCreateObjectInternal overflow attempt
9	12800	SHELLCODE base64 x86 NOOP
9	17344	SHELLCODE x86 OS agnostic xor dword decoder
7	15930	NETBIOS Microsoft Windows SMB malformed process ID high field remote code execution attempt
7	7035	NETBIOS SMB Trans mailslot heap overflow attempt
6	12802	SHELLCODE base64 x86 NOOP
5	15199	NETBIOS SMB NT Trans NT CREATE param_count underflow attempt
5	542	CHAT IRC nick change
4	2508	NETBIOS DCERPC NCACN-IP-TCP lsass DsRolerUpgradeDownlevelServer overflow attempt
4	11945	NETBIOS SMB Trans2 OPEN2 maximum param count overflow attempt
3	16417	NETBIOS SMB Negotiate Protocol Response overflow attempt
3	18555	MISC VERITAS NetBackup java authentication service format string exploit attempt
3	11289	RPC portmap mountd tcp zero-length payload denial of service attempt
2	17668	POLICY download of a PDF with embedded JavaScript - JS string
2	20670	SPECIFIC-THREATS Asterisk data length field overflow attempt
2	15264	WEB-CGI Oracle TimesTen In-Memory Database evtdump CGI module format string exploit attempt
1	12798	SHELLCODE base64 x86 NOOP
1	15255	ORACLE Secure Backup msgid 0x901 username field overflow attempt

Table 6.25: Predictions where the Machine Learning Agent's confidence is greater than 0.90 on TCP traffice excluding HTTP.

confidence values range from 0 to 1, the network defender can use their discretion to investigate traffic with slightly lower confidence values with the goal of finding malicious multi-agent system communication patterns that are different from those the misuse detection engine has identified, but still somewhat similar. The system guides the network defender's investigation, leading them from specific malicious instances to more general behaviour patterns indicative of malicious multi-agent system communications. However, the investigation is intended to be interactive (Section 4.4), so when the network defender encounters false positives, it is important that they either refine the misuse detection signatures that are associated with the false positives, or write misuse detection signatures to white list certain similar behaviours. By feeding back into the system the network defender influences the Multi-Agent Malicious Behaviour Detection systems capability to provide valuable guidance.

The experiment demonstrated that the Machine Learning Agent performed poorly when tasked with generalizing from specific attacks identified by misuse detection to more general attacks. Instead it tended to overgeneralize and produce far too many false positives. This makes a network defender crucial to analyze the results. Even so, the number of false positives makes the task daunting on a network that is under heavy attack, as in the case of the dataset used for this experiment. While looking at each false positive would be impossible for a network defender, the system provides a mechanism for the network defender to provide feedback into the learning by observing a subset of the false positives and writing misuse detection signatures to whitelist that traffic. Recall that no whitelisting was performed in this experiment, as was done in earlier experiments. This experiment highlights the importance of whitelisting well-known benign traffic.

Even taking into account the poor classification performance, the framework as a whole did achieve its aim. The system agents generated a variety of feature set types and shared that information with other agents in the system, enabling Machine

Learning Agents to make recommendations to network defenders. The co-operation between the agents was achieved successfully, and each network session was successfully matched to the misuse detection alert. Once the results where generated, I could map the feature set back to a set of packets to verify the results.

There are a number of reasons to explain the poor classification performance. In this experiment the system was not given a whitelist of traffic to classify as benign (as in the *classify all* mode discussed in Section 4.5.3). Instead the system could only classify each feature set as a malicious attack similar to what was previously observed. The confidence of the classification was intended to offset this by indicating to the network defender what classification were worth following up on. Additionally, given that the traffic was generated by the Breaking Point to emulated real world attacks, often each attack would occur less than five times. I've shown in the previous experiment that the Machine Learning Agents could identify traffic with a low number of instances. However, in the dataset provided for this experiment, it seems the sheer number of low instance occurrences encouraged the Machine Learning Agents to overgeneralize. Feature selection also plays an important role in classification. While the Machine Learning algorithms chosen have been shown in this research and other research[Saffari et al., 2010] to be capable of performing difficult classification tasks, the features chosen during this implementation are not diverse enough to enable reliable classification.

6.6 Summary

The experiments described in this chapter demonstrate the capability of the Multi-Agent Malicious Behaviour Detection implementation to process network traffic, derive features from traffic, share the traffic with other agents, and perform classification on the traffic to make recommendations to network defenders. The results show that the Multi-Agent Malicious Behaviour Detection implementation is capable of accurately identifying a variety of benign network traffic types, and is also capable of identifying some types of malicious traffic accurately. However, it also identifies a number of weaknesses in using machine learning for classifying network traffic, and those will be further discussed in Chapter 7. The next chapter will described how this research as a whole addresses the research questions, and will explore avenues of future work pointed to by the evaluation described in this chapter.

Chapter 7

Findings and Recommendations

7.1 Overview

The research described in this thesis was undertaken to answer the research questions in Chapter 1. However, during my work toward answering those questions, I have become aware that this research involves a wide variety of issues that were not completely framed in those research questions. Issues I have dealt with in the course of this research include:

- the difficulties of interacting with live malicious multi-agent systems,
- the limitations of various machine learning algorithms,
- the scope of feature selection,
- the difficulty of managing large amounts of false positives,
- and the complexity of human-machine interactions.

In this chapter I will begin by discussing my findings and relating those back to the research questions posed in Chapter 1. Following this, I will present the contributions that this work offers to the research community. Finally, I will discuss future work related to this research.

7.2 Findings and Analysis

Chapter 1 presented several research questions around which the work in this thesis is focused. I will reiterate those questions here and discuss the results in my design, implementation and experiment in light of these questions.

7.2.1 Research Question 1

How can advances in artificial intelligence, such as machine learning and multi-agent systems, be applied to computer security to increase the success of detecting and mitigating malicious multi-agent systems?

Machine learning has a role to play in supporting the detection of malicious multi-agent systems. However, this research has demonstrated some weaknesses in current machine learning algorithms that limit it to the role of an adviser to a trained network defender. This research highlights the potential for existing machine learning algorithms to classify and make recommendations that guide network defenders in their investigations. As techniques in machine learning advance further, some of the weaknesses (i.e. over-generalization and false positives), may be overcome by the algorithms themselves, and the Multi-Agent Malicious Behaviour Detection framework provides a methodology to evaluate those improvements.

This research has also demonstrated a viable methodology for distributing work between a group of collaborating agents, such that they achieve the shared goal of classifying traffic in cooperation with existing misuse detection. I have demonstrated that the multi-agent paradigm is applicable to network detection, and further provided a framework to support future research into multi-agent systems for network detection. I have exploited advances in network communications, such as AMQP, to support multi-agent communications.

To summarize, advances in artificial intelligence improve the accuracy with which those systems can support network defenders in their attempts to detect and mitigate malicious multi-agent systems. While machine learning is limited given the current state of the art, improved techniques can be incorporated into my framework as they are developed.

7.2.2 Research Question 2

Given that malicious multi-agent systems are expected to evolve, including changing their communications to adapt to target networks, how can advances in artificial intelligence, such as machine learning and multi-agent systems, be used to improve discovery of novel and/or evolving malicious multi-agent systems?

This research has shown that Machine Learning Agents and misuse detection engines classify traffic using fundamentally different techniques. As malicious multi-agent system communications change, the misuse detection engine signatures gradually stop alerting on the traffic. However, the Machine Learning Agents will continue to alert on traffic similar to the traffic that used previously triggered misuse detection alerts. The labelled feature sets provide the network defender with a place to look for the malicious multi-agent system communications that are no longer identified through misuse detection. While false positives are expected, the labelled traffic still provides a subset of the much larger body of network traffic that the network defender would have to consider in their investigation. The Multi-Agent Malicious Behaviour Detection system will not identify every malicious multi-agent system communication, and certainly such a system is beyond the capabilities of current machine learning or artificial intelligence techniques. However, by providing a technique to generalize from specific instances of malicious multi-agent system communications, to unknown ones, the Multi-Agent Malicious Behaviour Detection system provides a methodology to discover evolving malicious multi-agent system communications.

Developing the Multi-Agent Malicious Behaviour Detection system as a multi-agent approach enables network defenders to develop new techniques to adapt their detection to changing malware communications. As was shown in Chapter 6, multiple Machine Learning Agents, each with slightly different learning techniques, can be deployed into the system to support various learning that may have more success identifying some forms of malicious multi-agent system communication over others. Additionally, new Feature Source Agents can support investigations where a malicious multi-agent system communication is suspected to be mimicking a specific protocol, and new features are required to distinguish between the benign and malicious sessions (as with the HTTP Feature Source).

7.2.3 Research Question 3

Given that the complexity of computer network security requires some level of human expertise, how can a methodology involving a collection of semi-autonomous Multi-Agent Malicious Behaviour Detection Agents improve the capability of network defenders to detect and monitor sophisticated malicious multi-agent systems? How can such a system limit the cognitive load on the network defenders while still providing increasing value in malicious multi-agent system detection capability?

The Multi-Agent Malicious Behaviour Detection system provides a methodology that supports collaborative human-machine detection and monitoring of malicious multi-agent systems. The role of the Multi-Agent Malicious Behaviour Detection agents is focused on recommending traffic for further analysis, given that it resembles traffic identified by human network defenders as interesting (via signatures deployed to a misuse detection system). The results of the experiments in Chapter 6 show that the Multi-Agent Malicious Behaviour Detection can successfully identify malicious multi-agent system behaviour, but suffers from a surplus of false positives. A confidence value can alleviate some of the cognitive load on the network defender, by guiding him or her to traffic that is more likely to contain malicious behaviour, based on its resemblance to past malicious behaviour.

Note that the task of identifying and tracking malicious multi-agent systems is difficult for even a trained network defender. So, while the Multi-Agent Malicious Behaviour Detection implementation produces a number of false positives, it provides an indicator that saves the network defender from searching through a much larger amount of traffic. Consider that while searching for malicious multi-agent system communications in a broad collection of network traffic might seem like searching for a needle in a haystack, the Multi-Agent Malicious Behaviour Detection system effectively reduces the size of the haystack, and provides network defenders with a mechanism to further reduce the size of the haystack by writing whitelisting signatures to teach the Multi-Agent Malicious Behaviour Detection system to identify some benign traffic.

The important aspect of the human-machine interaction is the mechanism for network defenders to feedback into the learning of the Multi-Agent Malicious Behaviour Detection system. When a network defender writes a misuse detection signature for a traditional misuse detection engine, they are identifying specific traffic. When a network defender writes a misuse detection signature and deploys it to a network with a Multi-Agent Malicious Behaviour Detection system, the network defender will have influenced the learning of the Multi-Agent Malicious Behaviour Detection and will receive indicators from Multi-Agent Malicious Behaviour Detection system regarding what other traffic is similar to the traffic they have just written a misuse detection signature to detect.

I am not discouraged by the low accuracy shown in the results of the complete higher education scenario (Section 6.5). Instead, the focus should be on the increase in value the Multi-Agent Malicious Behaviour Detection system provides over just deploying a misuse detection system. The Multi-Agent Malicious Behaviour Detection system provides the capability to occasionally find malicious behaviour that misuse detection systems do not. However, having a strong association to misuse detection signatures, Multi-Agent Malicious Behaviour Detection system recommendations provide more context than traditional anomaly detection. The system both reduces the cognitive load of network defenders by guiding them to traffic that potentially contains malicious communications, and adds value in malicious multi-agent system detection capability by generalizing from specific misuse detection rules.

7.2.4 Research Question 4

What methods exist to ensure that as research in multi-agent systems and machine learning progress, those benefits are realized in computer network security and employed against malicious multi-agent system detection?

The Multi-Agent Malicious Behaviour Detection framework, presented as part of this research, ensures that a method exists for integrating new techniques for both multi-agent systems and machine learning. I have demonstrated that Machine Learning Agents can be extended from the provided agent designs that are implemented using C++ and C# classes. As machine learning algorithms are improved or developed, these can be incorporated into new Machine Learning Agents. Similarly, I ensured that agent communications were implemented in an Agent base class, enabling additional agent types to be introduced by extending the Agent base class. The ensures that agent prototypes can be rapidly developed and deployed into the existing Multi-Agent Malicious Behaviour Detection system.

As demonstrated in Chapter 5, the entire system was implemented with extenability in mind. Additionally, I have taken advantage of object-oriented practices to abstract away unnecessary details from researchers wishing to develop new agents.

Dividing tasks between agents (i.e. intercepting traffic, crafting features, learning, etc.) enables developers to focus on improving specific parts of the framework without having to know about the details of the other tasks.

7.3 Contributions

This research is of interest to many research communities, including machine learning, distributed artificial intelligence, user interfaces and human-machine interaction. The contributions include:

1. A framework for Multi-Agent Malicious Behaviour Detection, that aids network defenders in the detection, mitigation, and study of malicious multi-agent system.

2. An extendable Multi-Agent Malicious Behaviour Detection implementation, built from the ground up with object-oriented principles in mind, making it possible to develop new agents using a variety of different programming languages and integrating those new agents into a population of existing agent types.

3. A methodology that allows Multi-Agent Malicious Behaviour Detection Agents to interact with malicious multi-agent systems, broadening the system's exposure to malicious multi-agent system behaviours, increasing the likelihood of discovering novel behaviours.

4. A methodology that enables shared discovery of novel malicious multi-agent systems between network defenders and semi-autonomous agents, by providing a mechanism for the Multi-Agent Malicious Behaviour Detection system to

ground hypotheses to network defender expertise. Additionally, the system provides a feedback mechanism for human network defenders to communicate their discoveries back into the Multi-Agent Malicious Behaviour Detection system.

5. Encouragement of further work in machine learning in this area by demonstrating the value of machine learning algorithms in the context of computer network security, and providing a mechanism to integrate new machine learning algorithms into the existing system using Machine Learning Agents.

7.4 Future Work

There are a number of directions that future work in this area can profitably take. Much of this worked has focused on building an architecture to support research in multi-agent systems for computer security. Now that the architecture is available, it can be built on for more interesting works. Here I will present a number of possible extensions to this work that would be suitable for graduate, post-graduate, or even undergraduate work.

7.4.1 User Interaction Study

For this study I was the sole source of security analytics for the experimental trials. There was no feedback from other potential security analysts. I would have liked to perform a complete usability study for this system. The usability study would involve recruiting a pool of subjects with varying expertise in network security. Each analyst could be classified by their previous experience with security products such as Snort.

Criteria would be devised for measuring the capability of analysts to identify threats to the network, the cognitive load on the analysts as well as the impact a well devised user interface has on the analysts understanding of the environment. Analogous work on examining cognitive load on robotic teleoperators has been done [Wegner and Anderson, 2006], and perhaps similar metrics could be employed. This research would address the gap between the overall capability that intelligent multi-agent systems such as this one provide, with the usability of those technologies in the real world environment. Further emphasis would be put on the responsiveness and adaptability of the system. Currently my implementation can communicate intention and analysis through log files and the 3D representation of traffic in the network, while the analysts communicate their intentions and analytics by updating and adding misuse based signatures to the system in reaction to information gained from the system. A well-designed user interface can encourage easier interaction between the analyst and the agents in the system. A benefit of the interaction study would be the implementation of a user interface that is intuitive to a variety of security analysts. The user interface demonstrated here is sufficient for this research, but it could be improved. I would like to explore ways of improving the interactions between agents and security analysts, such as introducing a more immersible user interface that allows the analyst to feel more "in control". Traditional user interfaces often seem passive, and have a high latency - where the analyst ends up watching as the malicious agents attack the system as opposed to interacting with them. Depending on the scope of the study, this work would be suitable for both masters and doctoral theses. Candidates for the experiments could be taken from volunteers in undergraduate computer security courses, or from industry.

7.4.2 Live Traffic Study

In this work, I have discussed the collection of malware and the infection of a protected virtual network for evaluating security systems. There are a number of legal, ethical and time constraints preventing me from persuing those opportunities. However, it is an obvious next step for proving the value of this work. Once some of the constraints were worked out, malware collection would be possible in a number of ways. For example:

Honeypot Tactics: Many researchers have aimed to set up honeypots in order to attract malicious software. The honeypot is a monitored machine that is vulnerable to attack. Such vulnerabilities may include no firewall, easy to guess passwords, unpatched software, older operating systems, etc. Since there is a significant amount of automated victim discovery, a honeypot has the potential to be compromised by automated as well as supervised attacks. There are, however, several disadvantages to honeypots. First, a large number of attacks require user intervention to initiate. Many attacks are content delivery: with no one to access a document infected with malware, the honeypot may never be infected. Second, once a honeypot is infected it may participate in malicious behaviour. There are some ethical issues with respect to allowing a machine to participate in malicious behaviour that may cause damage to other systems. Third, honeypots require patience and the infecting malware may not be readily identified, requiring a significant amount of work for it to be effective.

Deliberate Self Infection: By keeping informed of the latest zero day attacks through multiple sources, it is possible to identify websites hosting infected documents. For example, if a specific zero day attack involves redirecting hosts to a website to download malicious software, and the website is reported in the analysis of the zero day attack, a researcher could browse to the website intentionally in order to become infected. This is a preferred method, since it requires no official request or involvement with anti-virus vendors, and it also does not require waiting for someone to attack a honeypot. It also has the additional advantage of allowing the research to pick and choose which infections to go after, giving the researcher additional control.

Anti-Virus Companies: Anti-virus companies maintain scores of malware samples for their own analysis. Zero Day malware may be difficult to obtain. However, samples of older malware are likely obtainable if formally requested for research purposes. There are a number of anti-virus vendors to which such requests could be made, including McAffee, Norton, AVG, Avast, Microsoft Defender, Panda AV, Kaspersky Anti-Virus, etc.

Cyber Underground: Finally, malware is obtainable by becoming familiar with the cyber underground. By trolling forums and reading documentation, a researcher can solicit malware samples from individuals who experiment with malware. There are also publicly available malware development tools, such as Metasploit, that allow customization of malware attacks to suit attacker needs. These tools can be used to modify malware for research purposes.

Once infected, the researcher could study the interactions between live malware and their virtual infrastructure. By extending the existing Multi-Agent Malicious Behaviour Detection system and evaluating its performance, this work would be suitable for graduate level projects, or even a masters thesis.

7.4.3 Abstract Behaviours

This work has focused on features of sessions. However, there is an opportunity for hierarchies of machine learning agents learning from more abstract features of network communications. Features derived from multiple sessions, especially from sessions of different types whose individual interactions seem benign, but when considered together betray a pattern of malicious behaviour. Some of the limitations to overcome are the potential explosion of available features when the features of several sessions must be combined in an exponential number of combinations, deciding when to combine features from different sessions, what sessions are most likely to improve the systems capability to detect malicious behaviour, and the automation of those decisions. It is sensible to consider combining the features of DNS and HTTP sessions, as malware may first use DNS to identify where to send data and then send it over HTTP. However, is it more efficient to try to identify the behaviours in isolation or together, i.e. independent beaconing behaviour and the data ex-filtration behaviour or a single behaviour identified as the two combined. This is a significant extension of the current Multi-Agent Malicious Behaviour Detection system and is suitable for a masters level thesis.

7.4.4 Broader Machine Learning Comparison

While I am confident that I have chosen some of the leading online machine learning algorithms to trial with the Machine Learning Agents, there is an opportunity to re-use this architecture to study emerging machine learning algorithms. The architecture provides the flexibility to plug in new Machine Learning Agents, and then compare their accuracy with other Machine Learning Agents. There are a number of machine learning algorithm libraries available today, and the capability of integrating language-agnostic agents enables the capability for relatively quick research studies in Machine Learning, for example, for projects for undergraduate students.

7.4.5 Dynamic Feature Selection

In this research, the selection of features for inclusion in each feature set was static once an agent was deployed. However, much research has been done in identifying ideal feature selection to balance classification accuracy vs efficiency (Section 3.7). One concept that I had considered as a logical extension to this work is the inclusion of agents that are capable of deriving new features and dropping less valuable features to improve accuracy while maintaining efficiency. While the machine learning algorithms are effectively working out the value of particular features for classifying traffic, including novel features automatically without outside influence from a Security Analyst could potentially benefit agents, making it possible for them to evolve. Initially identifying features is manually intensive, and as far as I have read, while automated mechanisms exist for identifying valuable features from a set of features, the task of choosing a set of possible features is still a manual process in network security. This extension is ideal for graduate level research, as it would require extensive research in identifying existing methods for deriving features from traffic autonomously.

7.4.6 Impact of Encryption on Classification

Encryption poses an issue with a number of network detection systems (Section 3.8). While I did not directly address the issue in this research, future work could evaluate the impact of encryption on both standard misuse detection engine and the Multi-Agent Malicious Behaviour Detection system. As mentioned in Section 3.8, some behaviours are likely to demonstrate similar observable effects whether the data communicated are encrypted or not, so while the misuse detection engine might not be able to detect the keywords it is looking for in a packet, the features of the communications (e.g. number of packets, average size of packets, ports used, frequency, etc.) might stay relatively stable. This work would be suitable for an undergraduate project in computer security.

7.4.7 Impact of Distributed Denial of Service

The experiments in Section 6.5 highlighted the impact of a possible distributed denial of service attack through DNS on the machine learning agents. This naturally leads to the question of how a malicious multi-agent system might purposely impact the Multi-Agent Malicious Behaviour Detection system by sending specific types of traffic, such as distributed denial of service attacks, in an attempt to make the system learn specific traffic patterns. The experiment showed that the Multi-Agent Malicious Behaviour Detection system learned to classify the majority of DNS as malicious. A further bit of research would consist of subjecting a deployed Multi-Agent Malicious Behaviour Detection system to several different types of distributed denial of service attacks and evaluating the impact on further classifications once the distributed denial of service is complete. Can a malicious multi-agent system impact the Multi-Agent Malicious Behaviour Detection system such that it is less likely to identify malicious multi-agent system communications at a later date?

7.4.8 Efficiency Improvements

The agents implemented in this system have been optimized to varying degrees. However, I recognize that a somewhat naive approach I have taken is to divide the work across individual agents as unique processes. While some of the agents employ dispatch queues to take advantage of multiple cores on a host machine, I believe there is a significant benefit to be gained by evaluating mechanisms to share host resources. With six core processors becoming commonplace, there is more and more to gain by coding agents to distribute their work evenly among the available cores. Efficiency could also be gained in terms of how agents communicate, evaluating what messages to drop when there is congestion, quality control, and other factors. It is important to perform more research in identifying the bottlenecks in the system in order to propose potential improvements. The system as it stands can evaluate the throughput of alerts, features and packets, so any improvements in efficiency should be immediately measurable, making it an ideal for undergraduate research.

Another avenue of improvement would be hardware cards designed for fast string matching, and how to best take advantage of those cards to improve the system throughput. Metrics to identify the overall cost per gigabyte of traffic processed and classified could help to justify further research in multi-agent systems such as this one for wider acceptance by the network security community.

One of the advantages of employing autonomous agents is mobility, and there is also significant future work in exploiting this ability. Agents could be designed to recognize when the machine currently hosting them does not have the resources to keep up with the agent's resource demands. By maintaining a unique identifier in the system, the agent could transfer its current state to a different machine, perhaps by querying other agents in the system for their host specifications and current load and choosing a suitable host to move to.

7.4.9 Trust Relationships Between Agents

There is a degree of implied trust across the agents in this research. While the Alert Source agent can provide a degree of ground truth for the Machine Learning Agents, when several Alert Source Agents are involved, there may be inconsistencies in the generated alerts. There is also the possibility that some agents produce either false positives or false negatives. My approach takes the view that the false positives and false negatives should work themselves out to some degree, as the machine learning algorithms involved have some built in resistance to mis-classifications. However, an interesting extension to this work would be exploring the trust relationships that can develop between agents. Alert Source Agents that consistently agree with each other and the well trained Machine Learning Agents could be considered more trustworthy then those that consistently disagree. This also adds the additional variable of *status quo*, where introducing an agent that is capable of identifying threats that other agents cannot may appear as untrustworthy if it consistently disagrees with other agents, even though it is in fact identifying real or potential threats. On the other side of the equation is the agent that effectively runs off the rails and begins misclassifying a large percentage of benign traffic as malicious. Finally, there is the potential for malicious intentions, where Agents are designed to deceive other agents in the system by classifying traffic as benign when it provides their designer with a particular advantage. Also, one can consider how the security analyst can incorporate their trust evaluations of agents. Security Analysts may ignore the results of some tools if they don't agree with the analysts intuition. Providing a mechanism to blend the human and automated trust evaluations is important for a system to perform in the real world reliably. That particular element could be especially difficult to evaluate. However, there are a number of studies related to trust in multi-agent systems that can provide a basis for expanding on this work. Depending on the depth of the study on trust relationships, this study could provide enough work for a masters or doctoral level dissertation.

7.4.10 Security of Agents

In addition to implied trust across the agents, there is also the assumption that malicious agents existing in the network will not attempt to attack the Multi-Agent Malicious Behaviour Detection agents. Future work could focus on securing the communications between agents in the system to ensure that malicious agents cannot exploit the system. Some security measures already exist. For instance, each agent must announce itself using a uuid. Those uuid's are generated randomly at the moment, but they could be assigned from a registry so that only registered agents can participate in the system. There is also the capability built into RabbitMQ to enable

SSL encryption. Since all messages shared between agents are routed through the AMQP server, encryption can be enforced for all communications. The source code itself could contain potential security risks. Additionally, the impact of introducing security measures could be evaluated. As communications are encrypted, one would suspect an increase in latency and a decrease in overall performance of the agents. However, as this research has not addressed security issues related to agents, such studies would be interesting future work.

7.5 Conclusion

It is my hope that the success of this research encourages more research in malicious multi-agent system detection, by providing a Multi-Agent Malicious Behaviour Detection framework and implementation that can be extended by professional researchers, graduate students and undergraduate students alike. The framework enables researchers in machine learning to evaluate their algorithms in a complex dynamic domain, computer network security. Further, by demonstrating the benefits of a multi-agent approach, and implementing robust multi-agent communications, this work provides a platform to explore more complex interactions between Multi-Agent Malicious Behaviour Detection Agents. I believe that intelligent agents still lack the capability to perform all of the tasks required by network defenders autonomously, requiring interactions with human network defenders. This research has demonstrated that improving interactions between network defenders and the Multi-Agent Malicious Behaviour Detection framework increases the detection capability of both participants. If intelligent agents are not going to be capable of performing complex tasks in the near future, then the interactions of human network defenders and intelligent agents will only become more important as a subject for future research. Additionally, given that malicious multi-agent systems will continue to become more sophisticated, it is important to frame the problem as competing multi-agent systems. If researchers do not continue to leverage artificial intelligence (i.e. machine learning, multi-agent systems, teleautonomy) to improve the capabilities of agents defending networks, then malicious multi-agent systems will continue to wreak havoc on unprotected networks, as malware writers have strong financial motivations to continue to improve their work.

Appendix A

Multi-Agent Malicious Behaviour Detection Implementation Documentation

A.1 Overview

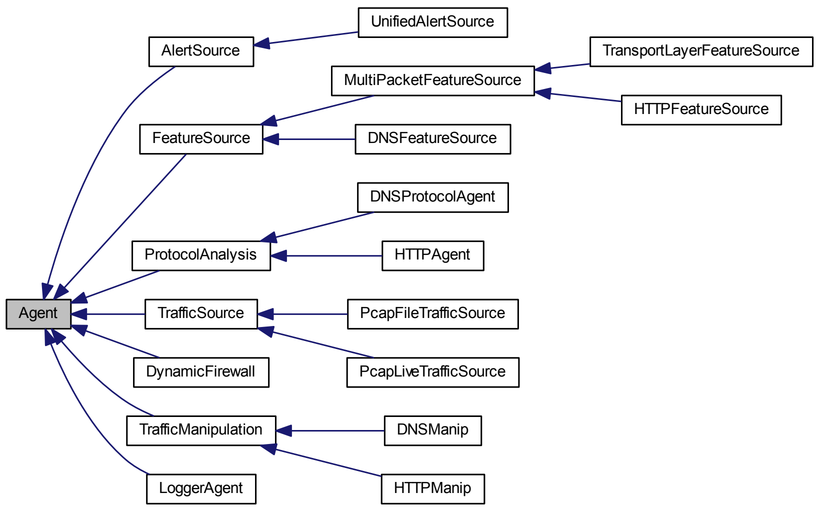
The following appendix provides the doxygen style documentation for the Multi-Agent Malicious Behaviour Detection implementation in Chapter 5. The documentation is provided here so that others can better replicate this work, and understand the details of the implementation choices further than is appropriate to present in the main body of the thesis. A motivating factor of this research is encouraging further work in Multi-Agent Malicious Behaviour Detection. This framework provides a potential starting point for researchers interested in exploring this research further.

A.2 Agent

A.2.1 Agent Class Reference

This is the abstract base class for all agents in the multi-agent malicious behavior detection system. This class provides the logic for connecting to and communicating with other agents in the system. It also sets out the interface for uniquely identifying agents, terminating agents graciously and interacting with the agent control channel.

Inheritance diagram for Agent:



Public Types

- enum **ControlState**

  A list of basic states that an agent can be in at any given time. Complex agents will maintain further state. However, each agent must at least be able to handle transition between the Control States.

Public Member Functions

- **Agent** (string host, int port)

  The base constructor for all agents. Sets up a connection to an AMQP server to communicate with other agents. Accepts a host and port number.

- virtual void **init** ()

  The initialization method should be called before any agent begins it's sense, plan and act cycle. Initializes the agent's ID as well as creating and binding to the agent control channel.

- abstract bool **sensePlanAct** ()

  *The agent's sense, plan, act cycle. This method should be invoked every cycle and represents the agent's capability to act autonomously. While there is no specific requirement that the agent performs each part of the cycle (sense, plan, and act), any class that extends the **Agent** (*p. *298) class should be able to operate autonomously by executing this method indefinitely.*

- virtual BasicDeliverEventArgs **checkControlChannel** ()

  Each agent polls the agent control channel for messages broadcast from other agents.

- virtual bool **getTerminate** ()

  Determine if the agent should launch termination routines.

- virtual bool **getComplete** ()

  Determine if the agent has successfully completed all prequisite termination routines.

- virtual void **onTerminate** ()

Provides each agent with an opportunity to perform any necessary task before termination. This method will send a message indicating that it is about to terminate and closes all communication channels.

- virtual void **onReset** ()

  Provides each agent with an opportunity to perform any necessary tasks in order to reset processing. After reset is completed the agent will return to a Set state, as though it has just been launched.

- virtual void **onPause** ()

  Provides each agent with an opportunity to perform any necessary task to suspend processing. After executing pause routines the agent should no longer process data. However, the agent should be capable of resuming processing from where is left off. Data received while paused should be ignored.

- virtual void **onStart** ()

  Provides each agent with an opportunity to perform any necessary tasks to enable processing. Should enable processing from either the Set or the Paused states.

- void **sendTerminate** ()

  Broadcast a message on the agent control channel indicating that all agents should terminate.

- virtual void **writeLog** (object source, ElapsedEventArgs e)

  Mechanism for producing basic log messages. Agents should re-implement the following method to produce more specific log messages. However, this method provides the basic information required for logging should logging be enabled. It will also send the log message to the appropriate exchange.

Public Attributes

- int **loglevel_**

  If logging is enabled, this value indicated the verbosity of the logs produced.

- string **logmessage_**

  Stores the current agent log information.

- string[ ] **ControlState_str** = { "Running", "Paused", "Set", "Terminated", "Complete" }

  A map of strings for the control states.

Protected Attributes

- IConnection **connection_**

  A connection to an AMQP server.

- string **AMQPHost_**

  The IP address or host name of the AMQP server.

- int **AMQPPort_**

  The port number that the AMQP server is listening on.

- string **agentType_**

  A two letter string indicating the type of agent.

- string **agentId_**

  A unique id for an agent. Used when interacting with other agents.

- **ControlState agentCntrlState_**

  The current control state of the ***Agent*** *(*p. *298).*

- System.Text.UTF8Encoding **encoding_**

  Instance of an encoder for writing messages and logs in UTF8.

Private Member Functions

- string **GenerateId** ()

  Generate a Unique 19 character alphanumeric ID for each agent. The id consists of 16 pseudo randomly chosen alphanumerics based on a guid concatenated with the two characters of the agent type.

Private Attributes

- ConnectionFactory **connectionFactory_**

  Connection factory to support AMQP messaging between agents.

- IModel **agentControlChannel_**

  A channel for broadcasting and receiving messages to all agents.

- QueuingBasicConsumer **agentControlConsumer_**

  A consumer for control messages broadcast to all agents.

- System.Timers.Timer **logTimer_**

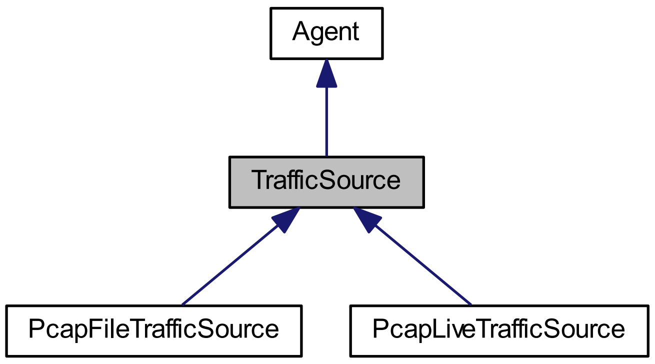
  If logging in enables, this timer indicates when a log message should be broadcast.

A.3 Traffic Source

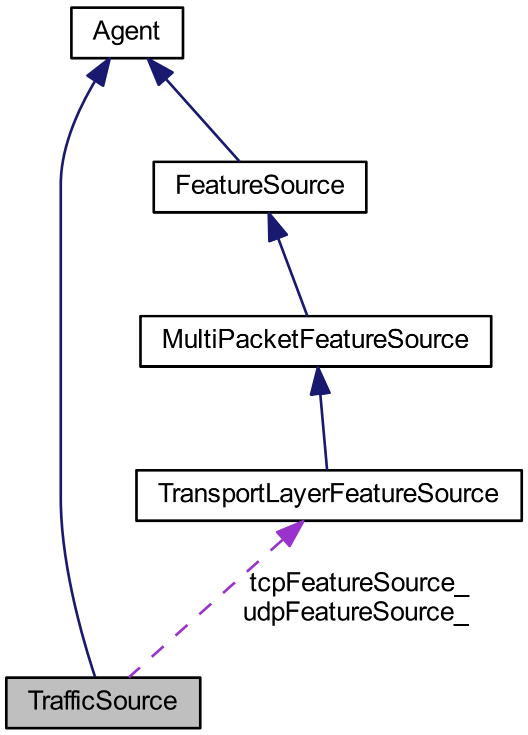
A.3.1 TrafficSource Class Reference

The **TrafficSource** (p. 303) abstract class is intended as a base class for objects implementing traffic source agents. It depends on the PacketDotNet C# libraries to provide an API for dealing with packet objects. The **TrafficSource** (p. 303) defines several AMQP connections in order to deliver messages to other agents in the system. Traffic source agents contain a traffic feature source by default, that is responsible for deriving features from the traffic processed by the traffic source agent.

Inheritance diagram for TrafficSource:



Collaboration diagram for TrafficSource:



Public Member Functions

- **TrafficSource** (string host, int port)

  Base constructor for any objects implementing a traffic source. ***Agent*** *(*p. *298) types that implement this abstract class will be identified as "TS" agent types. The constructor initializes a synchronized queue for managing captured packets and one or more feature source agents to derive features from the incoming packets.*

- override void **init** ()

Initialize the packet and packet summary exchanges on the target AMQP server for publishing messages. Also subscribe to the alert source exchange.

- override void **onTerminate** ()

  When instructed to terminate the traffic source will process any packets remaining in the packet queue, instruct the resident feature source agents to terminate, and finally call the base class on termination routine.

- override void **onReset** ()

  Clears out all of the existing structures, resets counters and instructs the resident feature source agents to perform their on reset routines. When finished, it will invoke the base class on reset routines.

- override void **onPause** ()

  Instruct any resident feature sources to pause and call the base class on pause routines.

- override void **onStart** ()

  Instruct any resident feature sources to begin processing and invoke the base class on start routines.

- override bool **sensePlanAct** ()

  The agent senses packets from the capture device, updates its internal state represented by one or more traffic features agents, and then acts by publishing messages for other agents to subscribe to.

- override BasicDeliverEventArgs **checkControlChannel** ()

  In addition to checking the agent control channel also ensure that the feature sources check their control channels.

- abstract bool **setupCaptureDevice** ()

  *Any agents extending the **TrafficSource** (*p. *303) must define a mechanism for setting up the capture device that will provide the packets for the source. When implemented, this method should access the capture device, perform configuration and start capturing packets.*

- string **getTopic** (Packet packet)

  Return the five tuple for this packet as an ASCII string topic. The topic consists of five elements; the server IP, the server port, the client IP, the client port and the transport protocol. All values should be hex strings, for example C4A80101.-0050.CAA80122.0400.0006. The server is assumed to be the side of the connection with the lowest port number.

- override void **writeLog** (object source, ElapsedEventArgs e)

  Add the number of packets published, the number of features labelled, and the number of features published to the log message. Then pass it on to the parent class to for transmission.

Public Attributes

- long **packetsPublished_**

  Number of packets published.

- long **packetSummaryPublished_**

  Number of packet summaries published.

- long **featuresPublished_**

  Number of features published.

Protected Member Functions

- void **receivePacket** (RawCapture rawPacket)

  TrafficSource *(*p. *303) implementations are intended to invoke this method to push packets into the packet queue. It abstracts away any requirement for the abstract class to understand how packets are generated (e.g. by reading a file or by capturing live traffic). The method expects packets formatted as PacketDotNet.-RawCapture objects. The packets include all packet data from the link layer on. Packets can be added to the packet queue asynchronously and are later processed synchronously.*

- bool **checkPktSumFilters** (byte[ ] packetSummary)

  Compare this packet against a set of software filters. If the packet summary passes the filter test this method returns true. Used primarily to identify when to publish packet summaries for other agents in the system. When bandwidth is limited filters are used to reduce the amount of traffic generated by the traffic source.

- bool **checkPktFilters** (byte[ ] packetSummary)

  Compare this packet against a set of software filters. If the packet summary passes the filter test this method returns true. Used primarily to identify when to publish packets for other agents in the system. When bandwidth is limited filters are used to reduce the amount of traffic generated by the traffic source.

Protected Attributes

- **TransportLayerFeatureSource tcpFeatureSource_**

  An instance of a Transport Layer Feature Source that maintains a set of features for TCP sessions observed by the agent.

- **TransportLayerFeatureSource udpFeatureSource_**

  An instance of a Transport Layer Feature Source that maintains a set of features for UDP sessions observed by the agent.

- Queue **packetQ**

  A queue of packets capable of queuing packets asynchronously and managing their flow synchronously, such that packets are processed one at a time.

- IModel **alertSourceChannel_**

  A channel for communications with alert source agents.

- QueuingBasicConsumer **alertSourceConsumer_**

  Basic Consumer to consume messages from alert source agents.

- IModel **packetChannel_**

  A channel for publishing packet data to subscribing agents.

- IModel **packetSummaryChannel_**

  A channel for publishing packet summaries to subscribing agents.

- ICaptureDevice **device**

  The device to capture packets on.

- LinkedList< **SoftPktFilter** > **pktExchFilters**

  List of filters to apply to packets destined for the packet exchange.

- LinkedList< **SoftPktFilter** > **pktSummaryExchFilters**

  List of filters to apply to packets destined for the packet summary exchange.

Private Member Functions

- byte[ ] **processPacket** (PosixTimeval time, Packet packet)

  Provide the feature source agent with access to a packet. This method acts primarily as a wrapper for the feature sources process packet method.

Private Attributes

- PosixTimeval **currentTime_**

  A timestamp from the most recently observed packet.

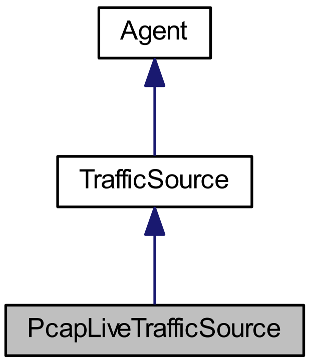
- int **udpfreq_**

  Number of iterations before checking if any udp session should expire.

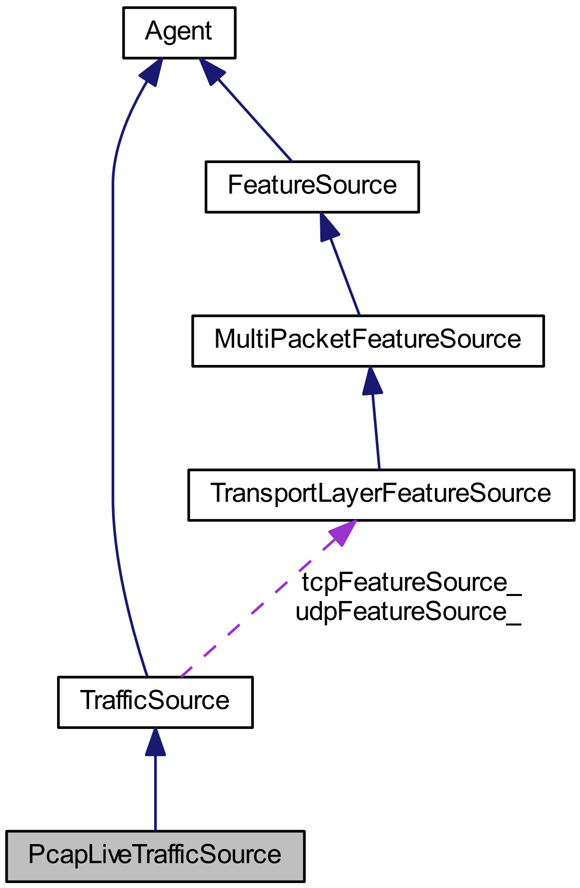
A.3.2 PcapLiveTrafficSource Class Reference

PcapLiveTrafficSource (p. 309) is an implementation of a traffic source agent that captures packets from an ethernet card in promiscious mode. The agent is intended to passively capture packets, preferably on a tap. It is not intended to operate inline.

Inheritance diagram for PcapLiveTrafficSource:



Collaboration diagram for PcapLiveTrafficSource:



Public Member Functions

- **PcapLiveTrafficSource** (int dev, string host, int port)

  Create a traffic source that passively captures packets from a live capture device. Devices on most machines are identified by a device ID. The constructor accepts a device ID but does not attempt to access the device.

- override void **onTerminate** ()

  When the ***PcapLiveTrafficSource*** *(*p. *309) is required to terminate, it closes the capture device, and calls the parent class on terminate routines.*

- override bool **sensePlanAct** ()

  The agent's sense, plan, act cycle. The cycle executes as long as the agent exists. The agent senses packets from the capture device, adjust it's internal state of traffic flows and then acts by publishing messages for other agents to subscribe to.

- override bool **setupCaptureDevice** ()

  Try to access the capture device using the device ID supplied at construction. If the device is accessed successfully then a callback method is supplied to handle the incoming packets. Packets will arrive asynchronously. However, as packets arrive they are added to a queue so that they can be processed synchronously.

Private Member Functions

- void **device_OnPacketArrival** (object sender, CaptureEventArgs args)

  Since packet arrivals are not predictable, push them into a synchronous queue so that the agent can process them when it has finished processing the previous packet. This is the callback method supplied to the device during the call to setupCapture-Device.

Private Attributes

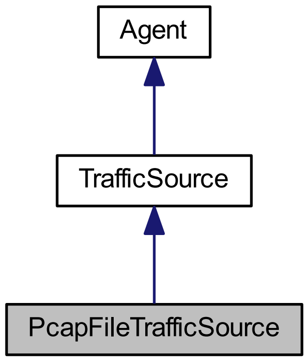
- int **deviceID**

  The network device identifier.

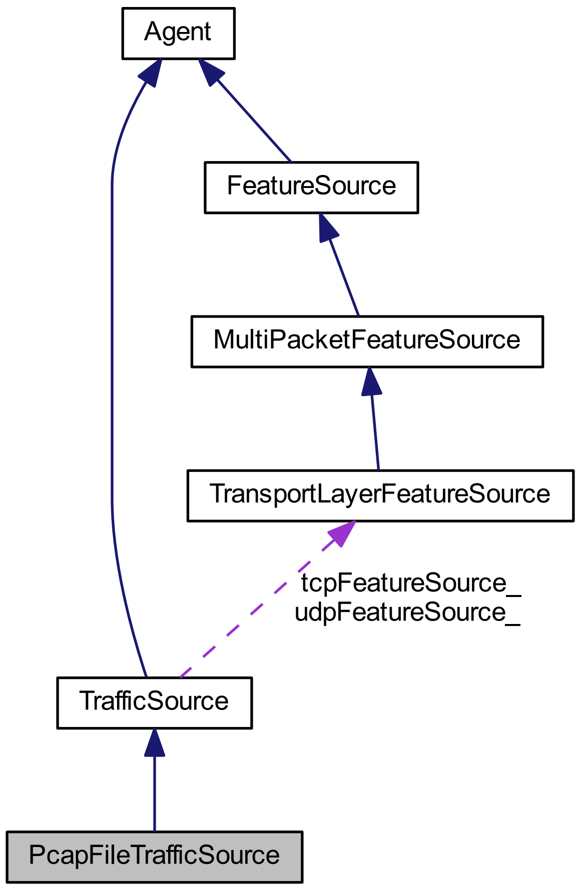
A.3.3 PcapFileTrafficSource Class Reference

PcapFileTrafficSource (p. 313) is an implementation of a traffic source agent that reads packets from pcap formatted files.

Inheritance diagram for PcapFileTrafficSource:



Collaboration diagram for PcapFileTrafficSource:



Public Member Functions

- **PcapFileTrafficSource** (string file, string host, int port)

  Create a traffic source that reads from a predefined packet capture file. The constructor requires the file name to read from.

- **PcapFileTrafficSource** (string file, double pktDelayMS, string host, int port)

Create a traffic source that reads from a predefined packet capture file. The constructor requires the file name to read from and a packet delay in milliseconds.

- override void **onTerminate** ()

  *On termination the **PcapFileTrafficSource** (*p. *313) closes its handle to the pcap file and calls the parent class termination routines.*

- override void **onReset** ()

  *On reset the **PcapFileTrafficSource** (*p. *313) closes its handle to the pcap file and then reopens the pcap file by calling setupCaptureDevice. The agent also invokes its parent class on reset routines.*

- override void **onPause** ()

  On pause the logging timer is stopped and the parent class on pause routines are called.

- override void **onStart** ()

  On start the logging timer is started and the parent class on start routines are called.

- override bool **sensePlanAct** ()

  The agent's sense, plan, act cycle. The cycle executes as long as the agent exists. The agent senses packets from the capture device, adjust it's internal state of traffic flows and then acts by publishing messages for other agents to subscribe to.

- override bool **setupCaptureDevice** ()

  Setup a packet source from a previously captured pcap file. For offline analysis. Especially useful when the various agents in the system are not capable of keeping *up with live traffic. Also enables repeatable experiments.*

- override void **writeLog** (object source, ElapsedEventArgs e)

  Override the writeLog mechanism to add in details of the current packets per second and megabits per second achieved by the pcap file traffic source. Also calls the parent classes writeLog.

Private Member Functions

- void **getNextPacket** (object source, ElapsedEventArgs e)

  A wrapper for managing packets read from the pcap file. This method can be used as an event handler to enforce a specific packet rate. Grabs a packet and passes it up to the parent class (TrafficSource) using the receivePacket method.

Private Attributes

- string **pcapfile_**

  Name of file containing pcap traffic capture.

- double **packetDelayMS_**

  Desired amount of time between packets read from the capture file.

- Stopwatch **st_**

  A timer for reading packets at the desired rate.

- double **bytesPerInterval_** = 0

  The Number of bytes processed this interval (used for logging).

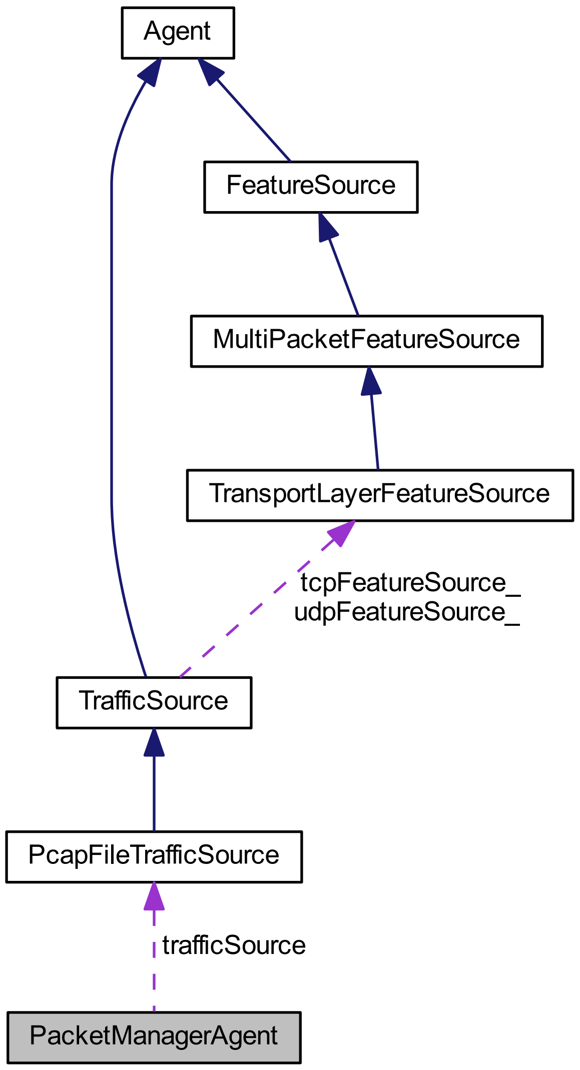
- double **packetPerInterval_** = 0

  The Number of packets processed this interval (used for logging).

A.3.4 PacketManagerAgent Class Reference

The Packet Manager Agent is standalone application that instantiates a traffic source agent.

Collaboration diagram for PacketManagerAgent:



Static Public Attributes

- static **PcapFileTrafficSource trafficSource**

  The traffic source instance.

Static Private Member Functions

- static void **Main** (string[ ] args)

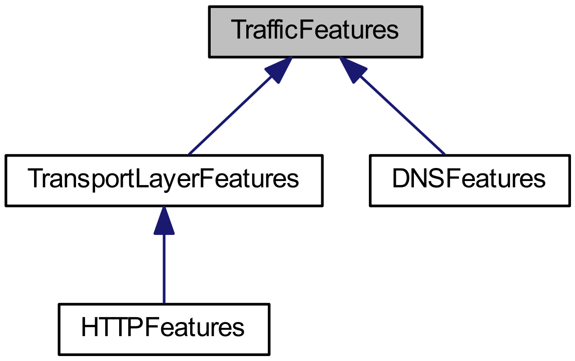
  Instantiate the traffic source instance and execute the sense, plan and act cycle until a termination request is received.

A.4 Feature Source

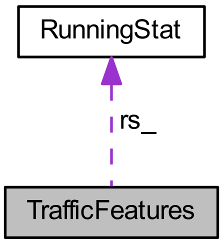
A.4.1 TrafficFeatures Class Reference

The **TrafficFeatures** (p. 318) object maintains a list of features for a given five tuple of traffic, where the five tuple is described by a server IP address, a server port, a client IP address, a client port and a protocol. As packets are passed into the **TrafficFeatures** (p. 318) object for a given five tuple, the features are updated to reflect the new state for this particular flow of traffic.

Inheritance diagram for TrafficFeatures:



Collaboration diagram for TrafficFeatures:



Public Member Functions

- **TrafficFeatures** (byte[ ] tuple)

  *The constructor for a **TrafficFeatures** (p. 318) object uses the passed in five tuple* *to initialize the object. The tuple is represented by an array of bytes where bytes 0,1,2 and 3 are the client IP, bytes 4 and 5 are the client port, bytes 6,7,8 and 9 are the server IP, byte 10 and 11 are server port and byte 12 and 13 are the protocol.*

- **TrafficFeatures** ()

  *Default Constructor for an empty **TrafficFeatures** (*p. *318) object.*

- byte[] **getkey** ()

  Get the TrafficFeature's 14 byte key, consisting of a 4 byte client IP address, a 2 byte client port, a 4 byte server IP address, a 2 byte server port and finally a 2 byte protocol value.

- virtual string **getTopic** ()

  *Return the five tuple for this **TrafficFeatures** (*p. *318) instance as a topic suitable for publishing to an AMQP server. The topic consists of five elements; the server IP, server port, client IP, client port and protocol. All values are represented as hex strings. For example, a Traffic Feature where the server IP is 196.168.1.1, the server port is 80, the client IP is 202.168.1.34, the client port is 1024 and the protocol is TCP will return the topic string C4A80101.0050.CAA80122.0400.0006.*

- virtual void **updateFeatures** (PosixTimeval time, Packet nextpacket)

  As packets are passed into the Traffic Features instance this method updates the features.

- virtual bool **finished** ()

  *A Traffic Feature by default does not contain logic to determine when it should expire aside from the timeout. Any classes extending **TrafficFeatures** (*p. *318)* *should override this methods if they have internal logic that indicates a closed connection (just as the FIN flag in TCP).*

- virtual string **toString** ()

  Creates a string representation of the ***TrafficFeatures*** *(*p. *318) object.*

- virtual byte[ ] **toSerializedData** ()

  Creates a serialized byte array representing the features of the ***TrafficFeatures*** *(*p. *318) object.*

- virtual void **deSerializeData** (byte[ ] serialized)

  Creates a ***TrafficFeatures*** *(*p. *318) object instance from a serialized byte array.*

Static Public Member Functions

- static byte[ ] **generateKey** (PacketDotNet.Packet packet)

  A static method that enables the generation of 14 byte keys by copying the five tuple out of a packet.

Public Attributes

- PacketDotNet.Packet **lastPacket_**

  The previous packet in the flow of packets. Used for some statistical calculations.

- long **lastPacketTime_**

  The time the last packet was received.

- System.Timers.Timer **featureTimer_**

  A timer that enables Traffic Features to age off if no packets are received for this five tuple after a fixed interval.

- UInt16 **featureType_**

  Any Traffic Feature that inherits from this class will set it's own feature type for serialization.

- UInt16 **protocol_** = 0

  The protocol of this traffic.

- UInt16 **label_** = 0

  A label assigned to this traffic.

- UInt16 **pred_** = 0

  A label assigned that may or may not agree with the original label.

- IPAddress **clientIP_**

  The ip address of the client.

- IPAddress **serverIP_**

  The ip address of the server.

- UInt16 **clientPort_** = 0

  The client port number.

- UInt16 **serverPort_** = 0

  The server port number.

- long **ticksSinceLastConnection_**

  The time since the last connection between the client and server participating in the traffic.

- PosixTimeval **starttimestamp_**

  Time stamp for the first packet associated with this traffic.

- PosixTimeval **endtimestamp_**

  Time stamp for the last packet associated with this traffic.

Protected Attributes

- byte[ ] **key_** = new byte[14]

  The key for this traffic feature object consists of a 4 byte client IP address, a 2 byte client port, a 4 byte server IP address, a 2 byte server port and a 2 byte value for the protocol. The key contains 14 bytes in total.

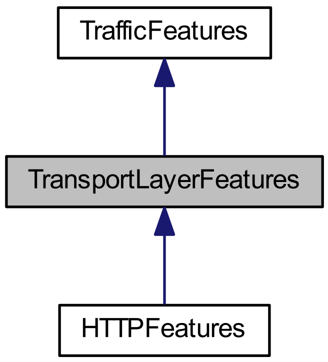
- **RunningStat rs_** = new **RunningStat**()

  Maintain a set of running statistics for this Traffic Features instance.

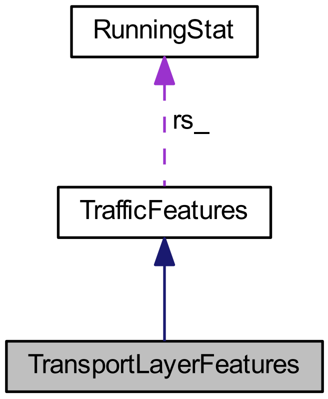
A.4.2 TransportLayerFeatures Class Reference

TransportLayerFeatures (p. 323) maintains a list of features for a given five tuple of traffic, extending on the features of the **TrafficFeatures** (p. 318) class. It includes additional features derived from the IP, TCP and UDP headers. Features are also maintained across several packets.

Inheritance diagram for TransportLayerFeatures:



Collaboration diagram for TransportLayerFeatures:



Public Member Functions

- **TransportLayerFeatures** ()

  Default Constructor for a Transport Layer Features object. It sets the feature type to 1 and calls the base class constructor.

- **TransportLayerFeatures** (byte[ ] tuple)

  Overloaded constructor for a ***TransportLayerFeatures*** *(*p. *323) object. It accepts a five tuple as a byte array for a target session. It sets the feature type to 1 and calls the base class constructor with the five tuple as an argument.*

- override void **updateFeatures** (PosixTimeval time, Packet nextpacket)

  As packets are passed into the ***TransportLayerFeatures*** *(*p. *323) instance this method updates the features.*

- override bool **finished** ()

  If both the client and server have sent packets with the TCP FIN flag set, return true to indicate that the connection is closed and the features should be published.

- override string **toString** ()

  Creates a string representation of the ***TransportLayerFeatures*** *(*p. *323) object.*

- override byte[ ] **toSerializedData** ()

  Creates a serialized byte array representing the features of the ***TransportLayerFeatures*** *(*p. *323) object. This method calls the base class's toSerializedData and then combines it with the serialized features of this features instance.*

- override void **deSerializeData** (byte[ ] serialized)

  Creates a ***TransportLayerFeatures*** *(*p. *323) object instance from a serialized byte array.*

Public Attributes

- const int **serialSize** = 116

  Number of bytes in the message when serialized.

- byte **maxDataControl_** = 0

  Maximum number of data control bytes observed in a packet during the lifetime of the traffic flow.

- byte **medDataControlAB_** = 0

  Median of data control bytes observed across all packets from client to server during the lifetime of the traffic flow.

- byte **q3DataControlAB_** = 0

  Third quartile of data control bytes observed across all packets from client to server during the lifetime of the traffic flow.

- ushort **maxDataOnWire_** = 0

  Maximum number of data bytes observed in a packet's Ethernet PDU during the lifetime of the traffic flow.

- ushort **maxDataWireBA_** = 0

  Maximum number of data bytes observed in a packet's Ethernet PDU from server to client during the lifetime of the traffic flow.

- ushort **maxDataWireAB_** = 0

  Maximum number of data bytes observed in a packet's Ethernet PDU from client to server during the lifetime of the traffic flow.

- ushort **reqSackBA_** = 0

If the end-point sent a SACK permitted option in the SYN packet opening the connection, reqSackBA_ is 1; otherwise it is 0 (server to client)

- ushort **maxSegmSizeAB_** = 0

  Maximum segment size observed from client to server during the lifetime of the traffic flow.

- ushort **maxSegmSizeBA_** = 0

  Maximum segment size observed from server to client during the lifetime of the traffic flow.

- ushort **minSegmSizeAB_** = ushort.MaxValue

  Minimum segment size observed from client to server during the lifetime of the traffic flow.

- ushort **minSegmSizeBA_** = ushort.MaxValue

  Minimum segment size observed from server to client during the lifetime of the traffic flow.

- ushort **maxDataIP_** = 0

  Maximum number of bytes observed in a packet's IP PDU during the lifetime of the traffic flow.

- ushort **maxDataIPAB_** = 0

  Maximum number of bytes observed in a packet's IP PDU from client to server during the lifetime of the traffic flow.

- ushort **maxDataIPBA_** = 0

  Maximum number of bytes observed in a packet's IP PDU from server to client during the lifetime of the traffic flow.

- ushort **medDataIPAB_** = 0

  Median of data bytes observed across all packet's IP PDU from client to server during the lifetime of the traffic flow.

- float **meanSegmSizeAB_** = 0

  Average segment size observed during the lifetime of the connection from client to server calculated as the value reported in the actual data bytes field divided by the actual data pkts reported.

- float **meanDataControlAB_** = 0

  Mean of data control bytes observed in packets from client to server during the lifetime of the traffic flow.

- double **varDataControlBA_** = 0

  Variance of data control bytes observed in packets from server to client during the lifetime of the traffic flow.

- int **totalSegms_** = 0

  Total segments observed during the lifetime of the traffic flow.

- int **totalPkts_** = 0

  Total packets observed during the lifetime of the traffic flow.

- int **totalPktsAB_** = 0

  Total packets observed from client to server during the lifetime of the traffic flow.

- int **totalPktsBA_** = 0

  Total packets observed from server to client during the lifetime of the traffic flow.

- LinkedList< ushort > **dataIPLengthsABList_** = new LinkedList<ushort>()

List of data IP PDU lengths observed from client to server.

- ushort[ ] **dclAB_** = new ushort[256]

  Total data control byte lengths from client to server.

- bool **clientFin_**

  Set to true when a packet from the client has the TCP FIN option set.

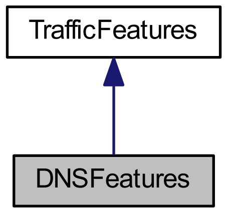
- bool **serverFin_**

  Set to true when a packet from the server has the TCP FIN option set.

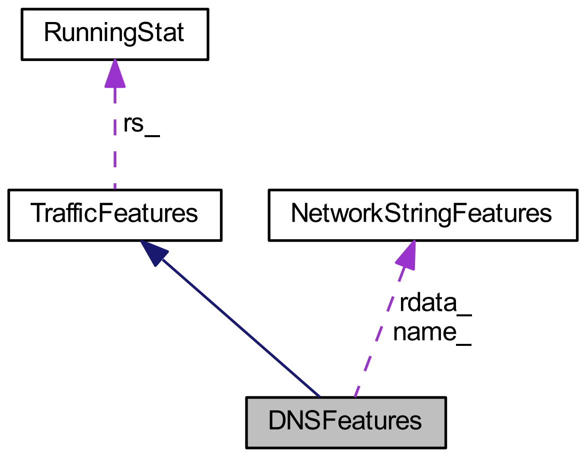
A.4.3 DNSFeatures Class Reference

The **DNSFeatures** (p. 329) object derives a list of features for a given DNS packet. It parses the entire DNS message, starting with the DNS header and then each DNS record, including all queries, answers, name servers and additional records. Combining the features of each record found in the message into a single set of features for the DNS packet ensures the feature sets for each DNS packet remains a consistent size.

Inheritance diagram for DNSFeatures:



Collaboration diagram for DNSFeatures:



Public Member Functions

- **DNSFeatures** (byte[ ] tuple)

  Overloaded constructor for a ***DNSFeatures*** *(*p. *329) object. Accepts an initial thirteen byte key. The constructor sets the feature type to 2 and passes the key to the base class constructor.*

- **DNSFeatures** ()

  Default Constructor for a ***DNSFeatures*** *(*p. *329) object. It sets the feature type to 2 and calls the base class constructor.*

- void **updateFeatures** (**DNSMessage** dnsMessage, PosixTimeval time, Packet nextpacket)

As packets are passed into the DNS Features instance this method updates the features. First the agent parses the DNS header, followed by all queries, answers, name servers and additional records. Network String Features objects are instantiated to track the features of the rdata and the name data. The base class (Traffic Features) updateFeatures method is also invoked.

- override void **updateFeatures** (PosixTimeval time, Packet nextpacket)

  Implements the base class update features method to ensure it adheres to the abstract class requirements.

- override string **getTopic** ()

  *Append the feature type "DNS" to the topic returned by a call the base class (-TrafficFeatures) getTopic method. With the addition of DNS feature type, the topic consists of six elements; the server IP, server port, client IP, client port, protocol and feature type. All values are represented as hex strings. For example, a **DNSFeatures** (*p. *329) instance where the server IP is 196.168.1.1, the server port is 80, the client IP is 202.168.1.34, the client port is 1024, the protocol is TCP and the feature type is DNS will return the topic string C4A80101.0050.CA-A80122.0400.0006.DNS.*

- override string **toString** ()

  *Creates and returns a string representation of the **DNSFeatures** (*p. *329) object.*

- override byte[ ] **toSerializedData** ()

  *Creates a serialized byte array representing the features of the **DNSFeatures** (*p. *329) object.*

- override void **deSerializeData** (byte[ ] serialized)

*Creates a **DNSFeatures** (p. 329) object instance from a serialized byte array.*

Public Attributes

- UInt16 **identifier_**

  Identifier created by the program to represent the query, used to match requests to replies.

- UInt16 **flags_**

  The flag bits including the bits aa, tc, rd, ra, and z.

- UInt16 **qCount_**

  The number of questions in the DNS message.

- UInt16 **aCount_**

  The number of answers in the DNS message.

- UInt16 **nsCount_**

  The number of name servers in the DNS message.

- UInt16 **arCount_**

  The number of additional records in the dns message.

- **NetworkStringFeatures rdata_**

  Features of the rdata message contained in the dns message.

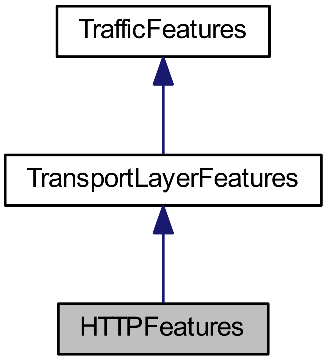
- **NetworkStringFeatures name_**

  Features of the name contained in the dns message.

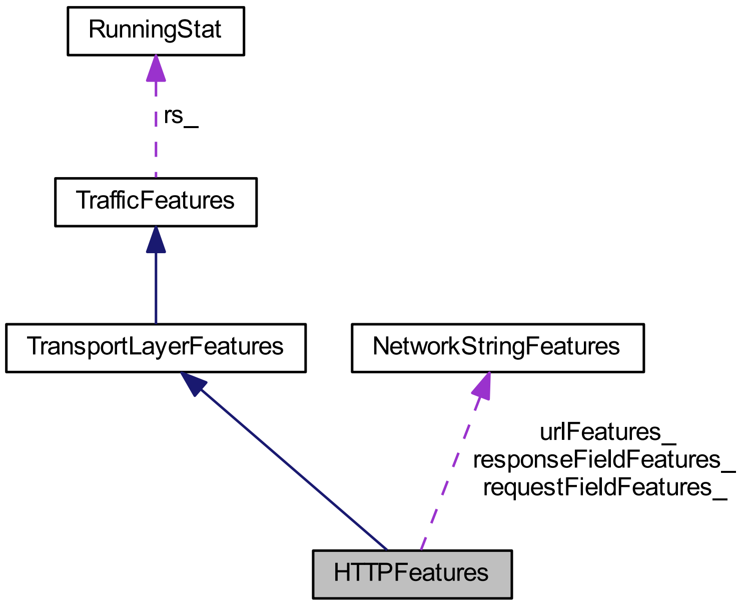
A.4.4 HTTPFeatures Class Reference

The **HTTPFeatures** (p. 333) object maintains a list of features for a given HTTP session. As new packets belonging to the given HTTP session are identified an HTTPFeature instance can update its features provided the next packet in the session.

Inheritance diagram for HTTPFeatures:



Collaboration diagram for HTTPFeatures:



Public Types

- enum **RequestFieldsEnum**

  An enumeration of the list of HTTP request header fields to extract features from.

Public Member Functions

- **HTTPFeatures** (byte[] tuple)

  Overloaded constructor for a ***HTTPFeatures*** *(*p. *333) object that accepts a five tuple. Sets the feature type to 3 and initializes place holders for a subset of request header fields and response header fields. Also initializes dictionaries for all parsed* *request and response header fields. Features for strings of interest are parsed by NetworkStringFeature objects. The five tuple provided to the constructor is passed to the base class constructor.*

- **HTTPFeatures** ()

  Constructor for a ***HTTPFeatures*** *(*p. *333) object. Sets the feature type to 3 and initializes place holders for a subset of request header fields and response header fields. Also initializes dictionaries for all parsed request and response header fields. Features for strings of interest are parsed by Network String Feature objects.*

- override void **updateFeatures** (PosixTimeval time, Packet nextpacket)

  As HTTP packets are passed into the ***HTTPFeatures*** *(*p. *333) instance, this method updates the features.*

- override string **getTopic** ()

  Append the feature type "HTTP" to the five tuple topic. The HTTP feature topic consists of five elements; the server IP, server port, client IP, client port, protocol and feature type. All values should be hex strings, except for the feature type. For example 000000AA.C4A80101.0050.CAA80122.0400.HTTP.

- int **extractHeaderField** (byte[ ] data, int sindex, out string fieldname, out byte[ ] fieldvalue)

  Traverse a line of an http response or request to extract a header field. It expects to see a "field name"0x3A0x20"value"0x130x12.

- int **extractMethod** (byte[ ] data)

  Traverse the first line of an http request to extract the method. It expects to see a "method"0x20"url"0x20"http version"0x130x12.

- override string **toString** ()

  Creates a string representation of the ***HTTPFeatures*** *(*p. *333) object.*

- void **prettyPrint** ()

  Creates a nicely formatted string that displays the session request and response headers, as well as a subset of the session features.

- override byte[ ] **toSerializedData** ()

  Creates a serialized byte array representing the features of the ***HTTPFeatures*** *(*p. *333) object.*

- override void **deSerializeData** (byte[ ] serialized)

  Creates an ***HTTPFeatures*** *(*p. *333) object instance from a serialized byte array.*

Private Types

- enum **ResponseFieldsEnum**

  An enumeration of the list of HTTP response header fields to extract features from.

Private Member Functions

- int **extractResponseStatus** (byte[ ] data)

  Traverse the first line of an http response to extract the response status. It expects to see a "method"0x20"url"0x20"http version"0x130x12.

Private Attributes

- string[ ] **HttpMethods** = {"GET","POS","HEA","PUT","DEL"}

  A list of HTTP methods to identify HTTP requests.

- string[ ] **HttpResponses** = { "HTTP/" }

  A list of strings found in HTTP responses to identify packets containing an HTTP response.

- string[ ] **RequestFields** = { "User-**Agent**", "Via", "Referer", "Host", "Cookie" }

  A list of HTTP request header fields to extract features from.

- string[ ] **ResponseFields** = { "Server", "Location", "Set-Cookie" }

  A list of HTTP response header fields to extract features from.

- string **method_**

  The method extracted from the HTTP request.

- string **url_**

  The URL extracted from the HTTP request.

- **NetworkStringFeatures urlFeatures_**

  The features of the URL extracted from the HTTP request.

- **NetworkStringFeatures**[ ] **requestFieldFeatures_**

  The features of a subset of the header fields extracted from the HTTP request.

- **NetworkStringFeatures**[ ] **responseFieldFeatures_**

  The features of a subset of the header fields extracted from the HTTP response.

- string **httpversion_**

  The HTTP version extracted from the HTTP request or the HTTP response.

- string **statusCode_**

  The status code (200, 401, etc...) extracted from the HTTP response.

- string **status_**

  The status extracted from the HTTP response.

- Dictionary< string, byte[]> **requestHeaderFields_**

  A collection of the header field names and values extracted from the HTTP request.

- Dictionary< string, byte[]> **responseHeaderFields_**

  A collection of the header field names and values extracted from the HTTP response.

- bool **requestHeader_** = false

  Indicates whether an HTTP request header was identified in the session.

- bool **responseHeader_** = false

  Indicates whether an HTTP response header was identified in the session.

- byte **numRequestHeaders_**

  The number of fields found in the HTTP request header.

- byte **numResponseHeaders_**

  The number of fields found in the HTTP response header.

A.4.5 NetworkStringFeatures Class Reference

The **NetworkStringFeatures** (p. 338) object represents the interesting features for the purposes of classifying the strings commonly found in protocols containing ascii encodings. Examples include DNS and HTTP. Given a string of bytes **NetworkStringFeatures** (p. 338) will determine the values of various features like the length, entropy and number of segments.

Public Member Functions

- **NetworkStringFeatures** ()

  Default constructor for creating a blank ***NetworkStringFeatures*** *(*p. *338) object. Primarily used to create a place holder for deserializing features sent across the network.*

- **NetworkStringFeatures** (byte[ ] str)

  Constructor that accepts an array of bytes representing a string found in a network protocol. The constructor extracts a subset of relevant features and stores them in this ***NetworkStringFeatures*** *(*p. *338) instance.*

- byte[ ] **toSerializedData** ()

  Creates a serialized byte array representing the features of the ***NetworkStringFeatures*** *(*p. *338) object.*

- void **deSerializeData** (byte[ ] serialized)

  Creates a ***NetworkStringFeatures*** *(*p. *338) object instance from a serialized byte array.*

Public Attributes

- const int **serialSize** = 22

  Number of bytes in the message when serialized.

- UInt16 **byteDistinct_**

  Number of distinct byte values in the string.

- byte **byteMinVal_**

Minimum byte value in the string.

- byte **byteMaxVal_**

  Maximum byte value in the string.

- UInt16 **asciiCap_**

  Number of ASCII capital letters (byte values 65-90) in the string.

- UInt16 **asciiLow_**

  Number of ASCII lower case letters (byte values 97-122) in the string.

- UInt16 **asciiDigit_**

  Number of ASCII digits (byte values 48-57) in the string.

- UInt16 **len_**

  Length of string in bytes.

- double **entropy_**

  The entropy of the string message.

- UInt16 **segments_**

  The number of segments the string contains (separated by "/",".",":",";" or "=").

A.4.6 FeatureLabel Class Reference

Class used to represent labels for sets of features, typically provided by Machine Learning Agents or Alert Source Agents. Since session keys only provide uniqueness within a specific period, this class provides the capability to track labels applied to the same session key over time, that may not represent a single session.

Public Member Functions

- **FeatureLabel** (byte[ ] key, ushort label, PosixTimeval tv)

  Create a Feature Label with a specific session key, label and timestamp.

- void **addLabel** (ushort label, PosixTimeval tv)

  Add a new label for a session matching this session key.

- ushort **getLabel** (PosixTimeval start, PosixTimeval end)

  Identify labels contained in the feature label list that are within a specific time range, and have the lowest label number. Typically lower labels are reserved for more specific classification (e.g. a label for Yahoo Mail should have a lower value then HTTP, since the former is more specific).

Public Attributes

- byte[ ] **key_**

  A 14-byte session key.

- List< KeyValuePair < PosixTimeval, ushort > > **labels_**

  A list containing pairs of labels and timestamps indicating when the label is valid.

A.4.7 RunningStat Class Reference

This class provides an implementation of a running stats mean, variance and standard deviation. This code is based on the code found at www.johndcook.-com/standard_deviation.html and based on work described in \citep{ling74stats} and \citep{chan83stats}. I have modified the source code by porting it to C# and adding comments. The technique involves calculating the statistical values over a moving window.

Public Member Functions

- **RunningStat** ()

  Constructor initializes the number of values pushed into the formula to 0 and returns a ***RunningStat*** *(*p. *341) object.*

- void **Clear** ()

  Clears the number of values pushed into the formula back to 0.

- void **Push** (double x)

  Push a value into the formula for calculating the running stats.

- int **NumDataValues** ()

  Return the number of data values pushed into the running stats formula.

- double **Mean** ()

  Calculates the current mean based on the running stats formula.

- double **Variance** ()

  Calculates the current variance based on the running stats formula.

- double **StandardDeviation** ()

  Calculates the current standard deviation based on the running stats formula.

Private Attributes

- int **m_n**

Number of values pushed into the running stats formula.

- double **m_oldM**

  The previous value of M from the running stats formula.

- double **m_newM**

  The current value of M from the running stats formula.

- double **m_oldS**

  The previous value of S from the running stats formula.

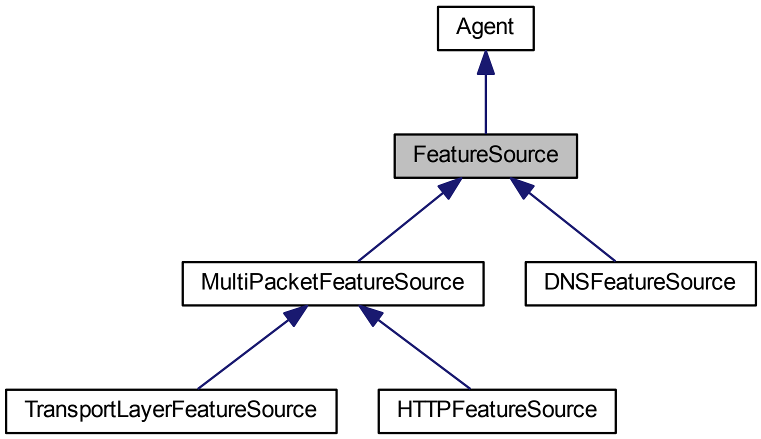
- double **m_newS**

  The current value of S from the running stats formula.

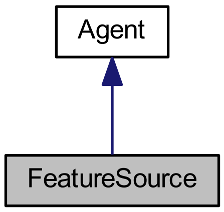
A.4.8 FeatureSource Class Reference

The **FeatureSource** (p. 343) abstract class is intended as a base class for objects implementing feature source agents. The **FeatureSource** (p. 343) defines a channel and queue for publishing features to other agents in the system. Feature source agents are responsible for deriving features from traffic passed to it from a traffic source.

Inheritance diagram for FeatureSource:



Collaboration diagram for FeatureSource:



Public Member Functions

- **FeatureSource** (string host, int port)

  *Base constructor for any objects implementing a **FeatureSource** (*p. *343). The* *constructor sets the agent type to "FS", creates a channel for features, and initializes a synchronized queue for managing features that are ready to publish.*

- virtual int **publishFeatures** ()

  Process the feature queue and publish all expired features out to subscribing agents.

- abstract void **add** (**TrafficFeatures** flow)

  *Abstract method, each agent implementing a **FeatureSource** (*p. *343) must provide a mechanism to add new features to the **FeatureSource** (*p. *343).*

- override void **onReset** ()

  On reset the feature source agent will clear out its current list of labels and expired feature sets. It will also reset counters. Finally the parent on reset is invoked.

- override void **onTerminate** ()

  Set the agents state to Complete and calls the parent on terminate routines.

Public Attributes

- Queue **syncExpiredFeaturesQ_**

  A Queue of feature objects that should be published and disposed of.

- Dictionary< byte[ ], **FeatureLabel** > **trafficFeatureLabels_**

  *A dictionary containing labels collected from an Alert Source **Agent** (*p. *298).*

- long **featuresPublished_**

  Counter to keep track of the number of features published.

- long **featuresLabelled_**

  Counter to keep track of the number of features labelled before being published.

Protected Attributes

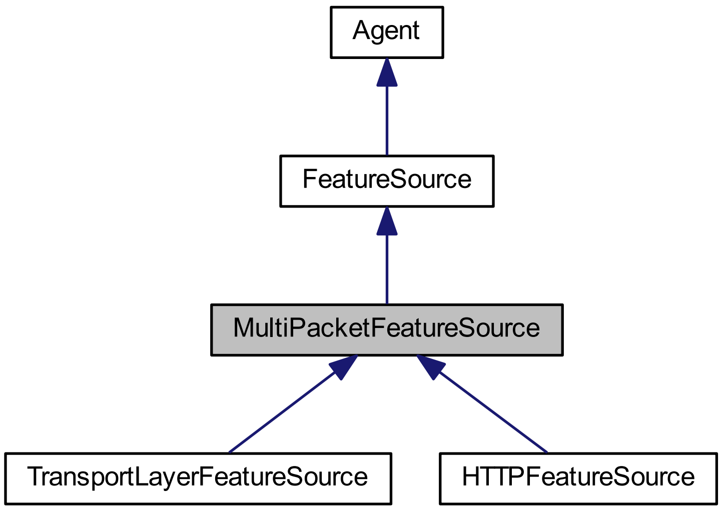
- IModel **featureChannel_**

  An AMQP channel for publishing features to subscribing agents.

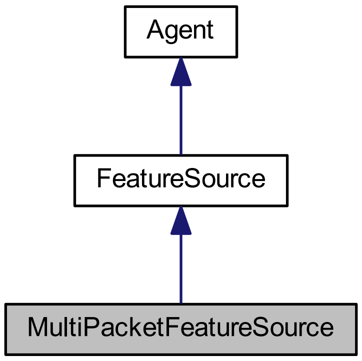
A.4.9 MultiPacketFeatureSource Class Reference

The **MultiPacketFeatureSource** (p. 346) extends the **FeatureSource** (p. 343) by providing the capability to maintain features across sessions of multiple packets.

Inheritance diagram for MultiPacketFeatureSource:



Collaboration diagram for MultiPacketFeatureSource:



Public Member Functions

- **MultiPacketFeatureSource** (string host, int port, PosixTimeval timeout)

  Constructor for a ***MultiPacketFeatureSource*** *(*p. *346). Initializes a structure to maintain the traffic features and the connection history structure. Also sets the timeout for feature sets.*

- override void **onTerminate** ()

  When instructed to terminate publish the remaining sessions in the traffic features structure. The remaining features will time out quickly since no more packets are processed after a terminate is received. Once all features are published the base on terminate routines are called.

- abstract byte[] **processPacket** (PosixTimeval time, Packet packet)

  Any agents implementing the ***MultiPacketFeatureSource*** *(*p. *346) must provide* *a mechanism to process packets passed to it by a traffic source. The implementation must include a time stamp, indicating when the packet was received and a copy of the full packet.*

- override void **add** (**TrafficFeatures** featureSet)

  Add a new traffic feature to the dictionary of features currently being tracked, indexed by the five tuple of the feature object. Also, update the connection history if this feature represents a session that is not currently being tracked.

- int **expireTrafficFeatures** (PosixTimeval curr, bool sendFeatures)

  Check existing feature sets against the time of the most recent intercepted packet. If the difference between the current packets timestamp and the most recent packet in the feature sets timestamp is greater then the timeout of the feature set place feature set into a synchronized queue for expired features. The method performs clean up necessary before features are published to other agents.

- PosixTimeval **timevalDiff** (PosixTimeval a, PosixTimeval b)

  Utility method for determining the time difference between two PosixTimeval objects.

- override void **onReset** ()

  On reset clear out the traffic features and connection history structures. Reset some internal counters. Call the parent classes on reset routines.

Static Public Member Functions

- static byte[ ] **normalizeKeyFromAlert** (byte[ ] buffer)

  Provided with a message from an alert source, this method will parse out and *normalize the five tuple contained inside it, providing a key that can be used to look up features.*

- static PosixTimeval **getTimeStampFromAlert** (byte[ ] buffer)

  Parse the timestamp from a Snort unified alert message. Used for matching alerts to sets of features.

Public Attributes

- ConcurrentDictionary< byte[ ], **TrafficFeatures** > **trafficFeatures_**

  A structure that contains the current traffic features indexed by five tuple keys.

- int **totalpackets_** = 0

  Total number of packets processed by the traffic flow feature tracker.

- PosixTimeval **timeout_**

  The timeout specifies how long the feature set should be tracked before it is published.

Protected Attributes

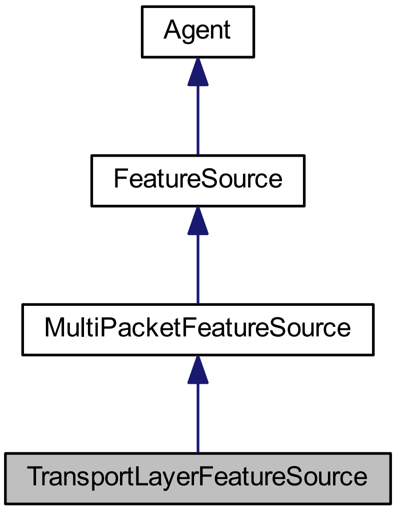
- Dictionary< byte[ ], DateTime > **connectionHistory_**

  A structure containing a history of all connections observed, indexed by five tuple key.

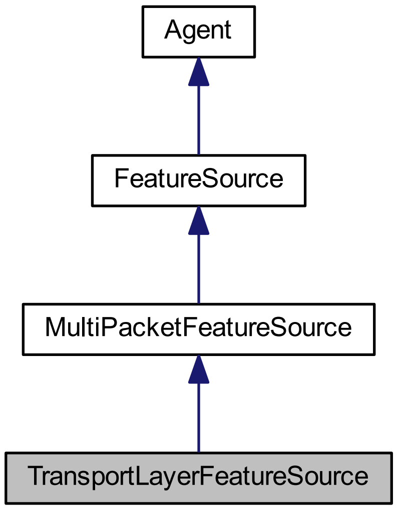
A.4.10 TransportLayerFeatureSource Class Reference

The **TransportLayerFeatureSource** (p. 349) is an implementation of a **MultiPacketFeatureSource** (p. 346) that manages multiple TCP and UDP feature sets.

Inheritance diagram for TransportLayerFeatureSource:



Collaboration diagram for TransportLayerFeatureSource:



Public Member Functions

- **TransportLayerFeatureSource** (string host, int port, PosixTimeval time-out)

  This constructor invokes the ***MultiPacketFeatureSource*** *(*p. *346) constructor.*

- override bool **sensePlanAct** ()

  The Transport Layer Feature source does not actively sense, plan or act. Instead it relies on a Traffic Source ***Agent*** *(*p. *298) to actively push in features that this agent will react appropriately to.*

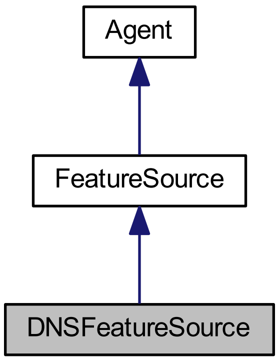
- override byte[] **processPacket** (PosixTimeval time, Packet packet)

Provides the agent with a mechanism to process packets passed to it by a traffic source. The time stamp indicates when the packet was received.

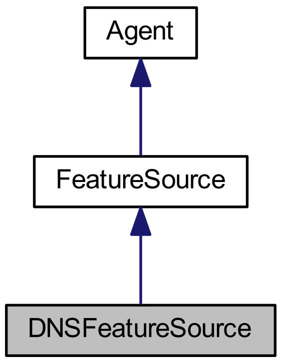
A.4.11 DNSFeatureSource Class Reference

The **DNSFeatureSource** (p. 352) implements the abstract class FeatureSource and is responsible for keeping track of multiple DNS features as the packets are passed in from another agent.

Inheritance diagram for DNSFeatureSource:



Collaboration diagram for DNSFeatureSource:



Public Member Functions

- **DNSFeatureSource** (string host, int port)

  This constructor calls the FeatureSource constructor. It does not perform any additional actions.

- override void **add** (**TrafficFeatures** feature)

  When traffic features are added to the DNS Feature Source they are directly inserted into the expired features queue in preperation for being published. The agent assumes that DNS packets consist of single packets, and ignores subsequent packets.

- override bool **sensePlanAct** ()

  The DNS Feature source does not actively sense, plan or act. Instead it relies on a Traffic Source Agent to actively push in features that this agent will react appropriately to.

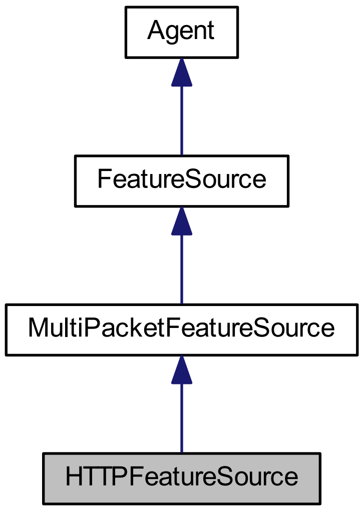
- override void **onTerminate** ()

  When instructed to terminate the agent will publish any features remaining in the expired features queue, before calling the base class on Terminate routines.

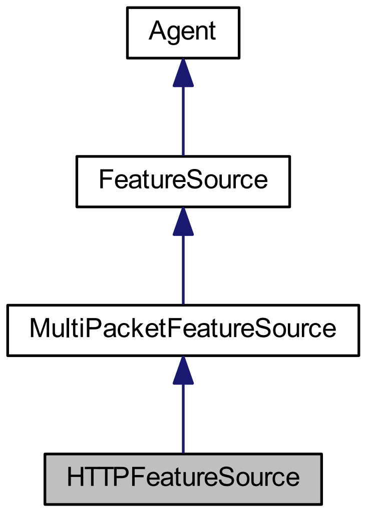
A.4.12 HTTPFeatureSource Class Reference

The **HTTPFeatureSource** (p. 354) implements the abstract class MultiPacket-FeatureSource and is responsible for keeping track of multiple **HTTPFeatures** (p. 333) objects as packets are passed in from another agent.

Inheritance diagram for HTTPFeatureSource:



Collaboration diagram for HTTPFeatureSource:



Public Member Functions

- **HTTPFeatureSource** (string host, int port, PosixTimeval timeout)

  This constructor calls the MultiPacketFeatureSource constructor. It does not perform any additional actions.

- override byte[] **processPacket** (PosixTimeval time, Packet packet)

  Process each packet as it's passed up from some traffic source. Each packet either adds an additional feature set for tracking or updates an existing set of features.

- override bool **sensePlanAct** ()

  The HTTP Feature source does not actively sense, plan or act. Instead it relies on a Traffic Source Agent to actively push in features that this agent will react *appropriately to.*

A.5 Machine Learning

A.5.1 MachineLearningAgent Class Reference

Implementation of an Alert Source Agent that classifies traffic using one of five online classifiers. The available machine learning algorithms are designed to operate on a stream of data samples, where the majority are unlabeled and require classification, while a subset are already labeled and used to evaluate and update the classification.

Inherits Agent.

Public Member Functions

- **MachineLearningAgent** (std::string configFilename)

  Construct a Machine Learning Agent instance. Load in the configuration of the Machine Learning Algorithm from the provided configuration file.

- ∼**MachineLearningAgent** ()

  Destroy all memory associated with the Machine Learning Agent.

- void **init** ()

  Initialize the Machine Learning Agent. Call the Agent constructor, build the classifier, and subscribe to AMQP channels.

- void **buildClassifier** ()

  Create a classifier capable of accepting a stream of data. The configuration file may contain a link to an initial training set of data or some attribute for a pre *trained classifier.*

- void **subscribeToFeatureSource** ()

  Subscribe to a feature source channel to consume feature source messages and apply classifications to the traffic feature sets that arrive on the channel.

- void **subscribeToLabelSource** ()

  Subscribe to a label source and consume labels from other alert source agents.

- void **onTerminate** ()

  Termination routines, launched when a termination message is received on the control channel.

- bool **getNextLabel** ()

  Check if any new labels have arrived.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextDataSample** ()

  Consume the next message available from a feature source.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextTrafficFeature-DataItem** (boost::shared_ptr< unsigned char > buf, int)

  When a traffic feature set arrives, process it and return a pointer to a Sample object, ready to send to the machine learning algorithm.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextDNSSample** (boost::shared_ptr< unsigned char > buf, int)

  When a DNS feature set arrives, process it and return a pointer to a Sample object, ready to send to the machine learning algorithm.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextHTTPSample** (boost::shared_ptr< unsigned char > buf, int)

When a HTTP feature set arrives, process it and return a pointer to a Sample object, ready to send to the machine learning algorithm.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextUDPSample** (boost::shared_ptr< unsigned char > buf, int)

  When a UDP feature set arrives, process it and return a pointer to a Sample object, ready to send to the machine learning algorithm.

- boost::shared_ptr < **TrafficFeaturesDataItem** > **getNextTCPSample** (boost::shared_ptr< unsigned char > buf, int)

  When a TCP feature set arrives, process it and return a pointer to a Sample object, ready to send to the machine learning algorithm.

- bool **sensePlanAct** ()

  The agents sense, plan and act cycle. Will repeat the loop as long as it is able. With each iteration it reads new messages from feature sources, updates it's internal state, makes plans based on the updated world model and then acts.

- void **writeSampleTraining** ()

  Write out sample training data and labels. The machine learning package requires a set of data to derive a series of characteristics for each machine learning algorithm. I don't train on the set produced here, but I do load it to setup the algorithm.

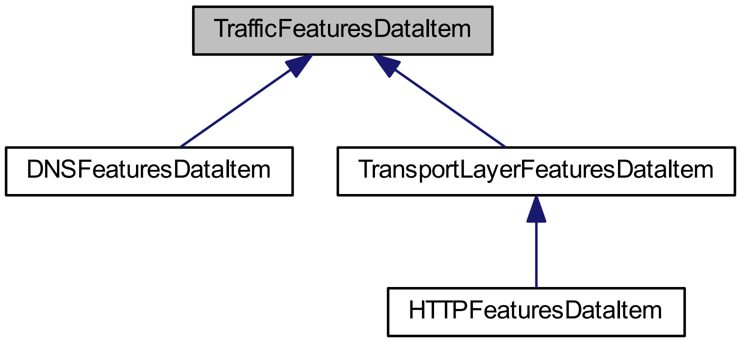
- void **writeLog** ()

  Write out a log describing the current training and testing accuracy achieved by the agent.

A.5.2 TrafficFeaturesDataItem Class Reference

A C++ implementation of the traffic features object's data members. This class is used to deserialize traffic flow features originating from a Feature Source into an object that agents implemented in C++ can manipulate.

Inheritance diagram for TrafficFeaturesDataItem:



Public Member Functions

- **TrafficFeaturesDataItem** (boost::shared_ptr< unsigned char > buffer, int features)

  Construct a traffic features data item from a buffer of bytes consumed from a feature source.

- ~**TrafficFeaturesDataItem** ()

  Destroy the feature source data item.

- std::string **toString** ()

Create a string representing the values for this traffic feature.

- Sample const & **getSample** () const

  Return a Sample object that was populated by the contents of this feature instance. The machine learning algorithms used in this research require data sets containing multiple Sample objects.

- void **populateSample** (const Hyperparameters &hp)

  Populate a Sample object from the contents of this traffic feature instance. The machine learning algorithms used in this research require data sets containing multiple Sample objects.

Public Attributes

- boost::shared_ptr< unsigned char > **featureBuffer_**

  The serialized feature, used when retransmitting is required.

- unsigned short **label_**

  A label assigned to this traffic.

- unsigned short **pred_**

  A class prediction for this feature.

- int **numfeatures_**

  The total number of features this object represents.

- unsigned short **featureType_**

  the type of feature.

- unsigned short **protocol_**

the protocol value from the IP header.

- unsigned int **clientIP_**

  The ip address of the client.

- unsigned int **serverIP_**

  The ip address of the server.

- unsigned short **clientPort_**

  The client port number.

- unsigned short **serverPort_**

  The server port number.

- long **ticksSinceLastConnection_**

  The time since the last connection between the client and server participating in the traffic flow.

- long **starttimestampS_**

  The time the first packet was received (seconds).

- long **starttimestampMS_**

  The time the first packet was received (milliseconds).

- long **endtimestampS_**

  The time the last packet was received (seconds).

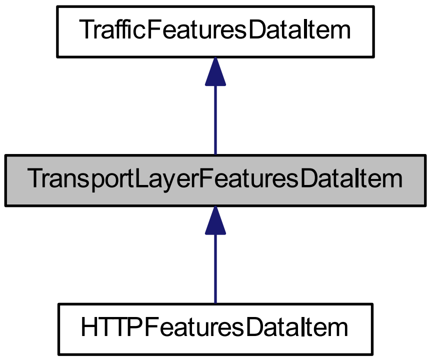
- long **endtimestampMS_**

  The time the last packet was received (milliseconds).

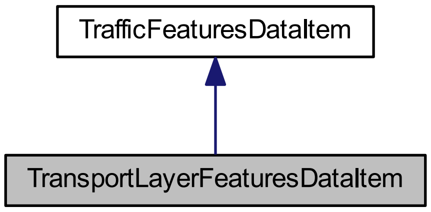
A.5.3 TransportLayerFeaturesDataItem Class Reference

A C++ implementation of the transport layer features object's data members. This class is used to deserialize transport layer features originating from a feature source into an object that agents implemented in C++ can manipulate.

Inheritance diagram for TransportLayerFeaturesDataItem:



Collaboration diagram for TransportLayerFeaturesDataItem:



Public Member Functions

- **TransportLayerFeaturesDataItem** (boost::shared_ptr< unsigned char > buffer, int features)

  Construct a transport layer features data item from a buffer of bytes consumed from a feature source.

- ∼**TransportLayerFeaturesDataItem** ()

  Destroy the transport layer features data item.

- void **populateSample** (const Hyperparameters &hp)

  Populate a Sample object from the contents of this DNS traffic feature instance. The machine learning algorithms used in this research require data sets containing multiple Sample objects.

- std::string **toString** ()

  Create a string representing the values for this traffic feature.

Public Attributes

- unsigned char **maxDataControl_**

  Maximum number of data control bytes observed in a packet during the lifetime of the traffic flow.

- unsigned char **medDataControlAB_**

  Median of data control bytes observed across all packets from client to server during the lifetime of the traffic flow.

- unsigned char **q3DataControlAB_**

Third quartile of data control bytes observed across all packets from client to server during the lifetime of the traffic flow.

- unsigned short **maxDataOnWire_**

  Maximum number of data bytes observed in a packet's Ethernet PDU during the lifetime of the traffic flow.

- unsigned short **maxDataWireBA_**

  Maximum number of data bytes observed in a packet's Ethernet PDU from server to client during the lifetime of the traffic flow.

- unsigned short **maxDataWireAB_**

  Maximum number of data bytes observed in a packet's Ethernet PDU from client to server during the lifetime of the traffic flow.

- unsigned short **reqSackBA_**

  If the end-point sent a SACK permitted option in the SYN packet opening the connection, reqSackBA_ is 1; otherwise it is 0 (server to client).

- unsigned short **maxSegmSizeAB_**

  Maximum segment size observed from client to server during the lifetime of the traffic flow.

- unsigned short **maxSegmSizeBA_**

  Maximum segment size observed from server to client during the lifetime of the traffic flow.

- unsigned short **minSegmSizeAB_**

  Minimum segment size observed from client to server during the lifetime of the traffic flow.

- unsigned short **minSegmSizeBA_**

  Minimum segment size observed from server to client during the lifetime of the traffic flow.

- unsigned short **maxDataIP_**

  Maximum number of bytes observed in a packet's IP PDU during the lifetime of the traffic flow.

- unsigned short **maxDataIPAB_**

  Maximum number of bytes observed in a packet's IP PDU from client to server during the lifetime of the traffic flow.

- unsigned short **maxDataIPBA_**

  Maximum number of bytes observed in a packet's IP PDU from server to client during the lifetime of the traffic flow.

- unsigned short **medDataIPAB_**

  Median of data bytes observed across all packet's IP PDU from client to server during the lifetime of the traffic flow.

- float **meanSegmSizeAB_**

  Average segment size observed during the lifetime of the connection from client to server calculated as the value reported in the actual data bytes field divided by the actual data pkts reported.

- float **meanDataControlAB_**

  Mean of data control bytes observed in packets from client to server during the lifetime of the traffic flow.

- double **varDataControlBA_**

Variance of data control bytes observed in packets from server to client during the lifetime of the traffic flow.

- int **totalPkts_**

  Total packets observed during the lifetime of the traffic flow.

- int **totalPktsAB_**

  Total packets observed from client to server during the lifetime of the traffic flow.

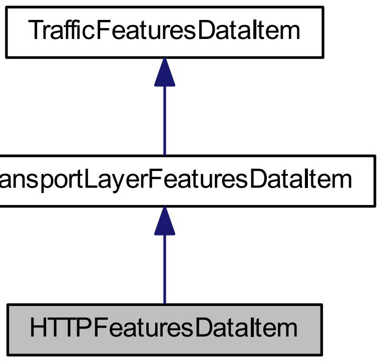
- int **totalPktsBA_**

  Total packets observed from server to client during the lifetime of the traffic flow.

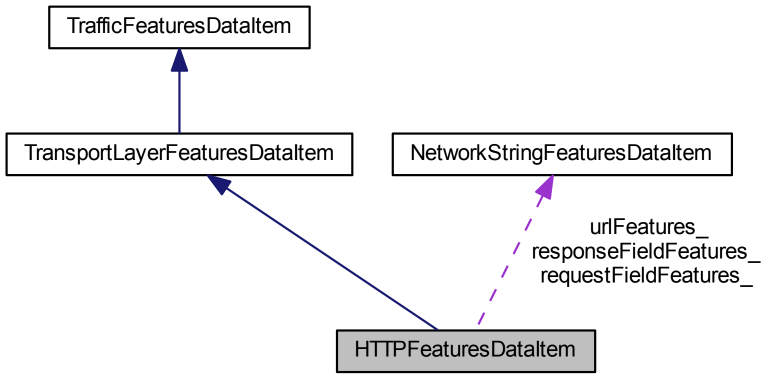
A.5.4 HTTPFeaturesDataItem Class Reference

A C++ implementation of the HTTP features object's data members. This class is used to deserialize HTTP features originating from a HTTP Feature Source into an object that agents implemented in C++ can manipulate.

Inheritance diagram for HTTPFeaturesDataItem:



Collaboration diagram for HTTPFeaturesDataItem:



Public Member Functions

- **HTTPFeaturesDataItem** (boost::shared_ptr< unsigned char > buffer, int features)

  Construct a HTTP features data item from a buffer of bytes consumed from a feature source.

- ~**HTTPFeaturesDataItem** ()

  Destroy the traffic flow data item.

- void **populateSample** (const Hyperparameters &hp)

  Populate a Sample object from the contents of this DNS traffic feature instance. The machince learning algorithms used in this research require data sets containing multiple Sample objects.

- std::string **toString** ()

Create a string representing the values for this traffic feature.

Public Attributes

- unsigned char **numRequestHeaders_**

  The number of header fields in the http request.

- unsigned char **numResponseHeaders_**

  The number of header fields in the http response.

- **NetworkStringFeaturesDataItem urlFeatures_**

  Features of the url in the http request.

- **NetworkStringFeaturesDataItem requestFieldFeatures_** [5]

  A list of interesting request header field features (User-Agent, Via, Referer, Host and Cookie).

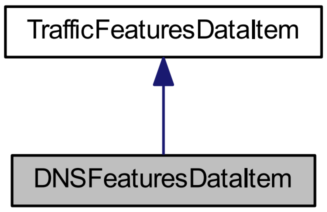
- **NetworkStringFeaturesDataItem responseFieldFeatures_** [3]

  A list of interesting response header field features (Server, Location and Set--Cookie).

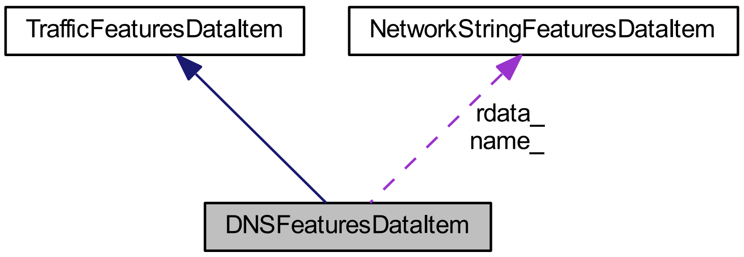
A.5.5 DNSFeaturesDataItem Class Reference

A C++ implementation of the DNS features object's data members. This class is used to deserialize DNS features originating from a DNS Feature Source into an object that agents implemented in C++ can manipulate.

Inheritance diagram for DNSFeaturesDataItem:



Collaboration diagram for DNSFeaturesDataItem:



Public Member Functions

- **DNSFeaturesDataItem** (boost::shared_ptr< unsigned char > buffer, int features)

  Construct a DNS features data item from a buffer of bytes consumed from a feature source.

- **~DNSFeaturesDataItem** ()

  Destroy the DNS features data item.

- void **populateSample** (const Hyperparameters &hp)

  Populate a Sample object from the contents of this DNS traffic feature instance. The machine learning algorithms used in this research require data sets containing multiple Sample objects.

- std::string **toString** ()

  Create a string representing the values for this traffic feature.

Public Attributes

- unsigned short **identifier_**

  Identifier created by the program to represent the query, used to match requests to replies.

- unsigned short **flags_**

  The flag bits including the bits aa, tc, rd, ra, and z.

- unsigned short **qCount_**

  The number of questions in the dns message.

- unsigned short **aCount_**

  The number of answers in the dns message.

- unsigned short **nsCount_**

  The number of name servers in the dns message.

- unsigned short **arCount_**

The number of answer records in the dns message.

- **NetworkStringFeaturesDataItem rdata_**

  Features of the rdata message contained in the dns message.

- **NetworkStringFeaturesDataItem name_**

  Features of the name contained in the dns message.

A.5.6 NetworkStringFeaturesDataItem Class Reference

A C++ implementation of the network string features object's data members. This class is used to deserialize Network String features originating from a feature source into an object that agents implemented in C++ can manipulate.

Public Member Functions

- **NetworkStringFeaturesDataItem** ()

  Default Constructor of a network string features to give it sensible default values.

- **NetworkStringFeaturesDataItem** (unsigned char ∗buffer)

  Construct a network string features data item from a buffer of bytes consumed from a feature source.

- ∼**NetworkStringFeaturesDataItem** ()

  Destroy the network string features data item.

- std::string **toString** ()

  Create a string representing the values for this traffic feature.

Public Attributes

- unsigned short **byteDistinct_**

  Number of distinct byte values in the string.

- unsigned char **byteMinVal_**

  Minimum byte value in the string.

- unsigned char **byteMaxVal_**

  Maximum byte value in the string.

- unsigned short **asciiCap_**

  Number of ASCII capital letters (byte values 65-90) in the string.

- unsigned short **asciiLow_**

  Number of ASCII lower case letters (byte values 48-57) in the string.

- unsigned short **asciiDigit_**

  Number of ASCII digits (byte values 48-57) in the string.

- unsigned short **len_**

  Length of string in bytes.

- unsigned short **segments_**

  The number of segments the string contains (separated by "/",".",":",";" or "=").

- double **entropy_**

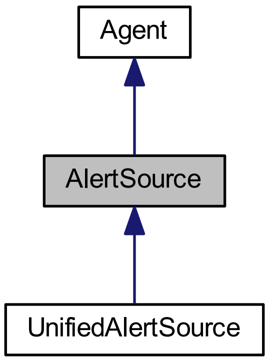
  The entropy of the string message.

A.6 Alert Source

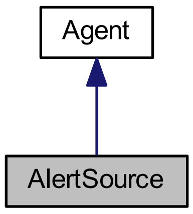
A.6.1 AlertSource Class Reference

The abstract **AlertSource** (p. 373) is intended as the base class for agents designed to publish alerts for subscribing agents in the system. This class provides a channel and queue for publishing the alert messages. AlertSources publish alerts using five tuple topics. The topics consist of a signature ID, the server IP address, the server port, the client IP address and the client port. Agents subscribe to alerts they are interested in using wildcarded topics. For example, an agent interested in all alerts from IP address 10.2.64.111 to 192.168.1.1 would subscribe to .0A02406F.∗.-C0A80101.∗.

Inheritance diagram for AlertSource:



Collaboration diagram for AlertSource:



Public Member Functions

- **AlertSource** (string host, int port)

  *Constructor for an **AlertSource** (*p. *373). Sets the agent type to "AS" and calls the base class constructor.*

- override void **init** ()

  Initialize an AMQP channel and create a queue for publishing alert messages. Also invokes the parent class init method.

- override void **onTerminate** ()

  Close all communication channels and call the base class on terminate routines.

- override void **onReset** ()

  Resets some internal counters and calls the parent on reset routines.

- override void **writeLog** (object source, ElapsedEventArgs e)

  Appends some additional information to the log message and then calls the parent's writeLog routines.

Protected Member Functions

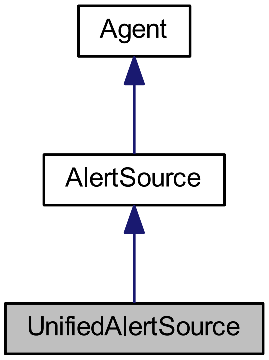
- abstract string **getTopic** (byte[ ] buffer)

  *Any agent implementing **AlertSource** (*p. *373) must provide a mechanism for generating a topic string from an alert message for publishing this alert to an amqp server. The topic should consist of five elements; the alert ID, the server IP, server port, client IP and client port. All values should be hex strings, for example 000000AA.C4A80101.0050.CAA80122.0400.*

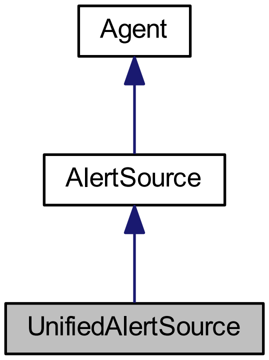
A.6.2 UnifiedAlertSource Class Reference

The **UnifiedAlertSource** (p. 375) is an implementation of an alert source agent. It is designed to read Snort unified alert files and publish the alerts for subscription by other agents in the system. The **UnifiedAlertSource** (p. 375) publishes alerts using five tuple topics. The topics consist of the signature ID, the source IP address, the source port, the destination IP address and the destination port. Agents subscribe to alerts they are interested in using wildcarded topics. For example, an agent interested in all alerts from IP address 10.2.64.111 to 192.168.1.1 would subscribe to .0A02406-F.∗.C0A80101.∗.

Inheritance diagram for UnifiedAlertSource:



Collaboration diagram for UnifiedAlertSource:



Public Member Functions

- **UnifiedAlertSource** (String alertFile, string host, int port)

*Constructor for a **UnifiedAlertSource** (*p. *375). Initialize the internal state to "Constructed", and call the parent class constructor.*

- override void **init** ()

  Initialize attempts to open a file stream where alerts are being written. Sets the state to "Initialized" if successful and "Terminate" if the the agent fails to open the file stream. Also calls the parent class init method.

- void **readHeader** ()

  Attempt to read the 16 byte unified alert file header. If the stream is smaller then 16 bytes, the agent sets the state to initialized and tries again, assuming the file header has not been written to the file yet. If less then 16 bytes are read, the agent sets its state to initialized and will try again. If 16 bytes are read but they do not contain the proper unified alert header bytes the agent sets the state to BadFormat and halts execution. Finally, if 16 bytes are read and the header appears to be in the right format, the state advances to ReadAlert.

- void **readAlert** ()

  Attempt to read 64 bytes of data from the file stream. When in the state ReadAlert, the agent assumes that the file pointer is at the beginning of the next alert to be read. If 64 bytes are read, the state is advanced to Publish. If less then 64 bytes are read, the agent resets the file stream point to where the alert record should be and maintains the state of ReadAlert (so the agent will try to read the alert again). If an exception is thrown during the attempted read the agent's state changes to BadFormat.

- void **publishAlert** ()

  Generate a topic from the 64 byte unified alert message and then publish the 64 *bytes as a message to the Alert exchange. If successful the agent's state is set back to ReadAlert. If any exception occurs the agent's state is set to BadFormat.*

- override BasicDeliverEventArgs **checkControlChannel** ()

  Overrides the standard check control channel so that the state of the agent can be changed depending on the result of any message read from the agent control channel.

- override void **onTerminate** ()

  The unified alert agent's termination routine. Closes the file stream and calls the parent class termination routines.

- override bool **getTerminate** ()

  Override the getTerminate method to base its decision on the agent's state instead of the terminate_ class member inherited from the parent class. Ensures the state machine handles all of the agent's transitions.

- override void **onReset** ()

  When instructed to Reset the agent will close the unified alert file stream and reopen it from the start, setting its state to "Initialized".

- override bool **sensePlanAct** ()

  The sense, plan, act cycle. In this case it defines the finite state machine, performing one action and checking the agent control channel for each cycle. When running correctly the agent should transition from "Constructed" to "Initialized", and then loop between "ReadAlert" and "Publish". A transition to "BadFormat" will terminate execution.

- byte[ ] **normalizeKeyFromAlert** (byte[ ] buffer)

Extract a key from a unified alert message.

Public Attributes

- int **blacklisted_**

  A count of how many alerts were ignored.

Protected Member Functions

- override string **getTopic** (byte[ ] buffer)

  Parse a Snort unified alert message and create a topic for publishing this alert to an AMQP server. The topic consists of five elements; the source IP, source port, destination IP and destination port and protocol. All values should be hex strings, for example 000000AA.C4A80101.0050.CAA80122.0400.

Private Attributes

- UASState **state_**

  The current state of the alert source agent, as the agent is implemented as a finite state machine.

- int **bytesread_** = 0

  The number of bytes read in the last read attempt.

- byte[ ] **buffer_** = new byte[64]

  A buffer of bytes to store unified alert message.

- FileStream **fileStream_**

  The stream where the alerts are being written.

- string **filename_**

  The file name of the stream where alerts are being written.

- uint[] **blacklist_**

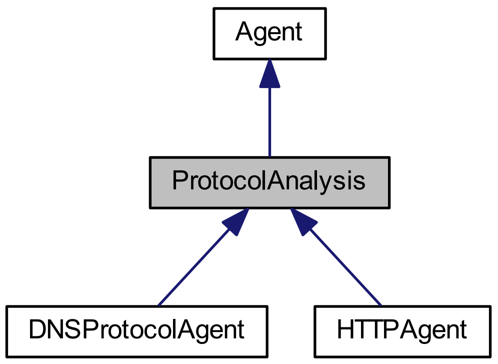
  An optional array of signature ids to ignore.

A.7 Protocol Analysis

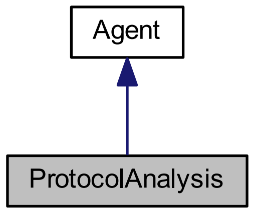
A.7.1 ProtocolAnalysis Class Reference

The **ProtocolAnalysis** (p. 380) abstract class is intended as a base class for objects implementing Protocol Analysis Agents. It depends on the PacketDotNet C# libraries to provide an API for dealing with packet objects. **ProtocolAnalysis** (p. 380) defines a source channel to accept incoming messages from a traffic source agent, and a request channel to accept requests from other agents. Each Protocol Analysis **Agent** (p. 298) must implement a process packet method to accept packets from Traffic Source Agents.

Inheritance diagram for ProtocolAnalysis:



Collaboration diagram for ProtocolAnalysis:



Public Member Functions

- **ProtocolAnalysis** (string host, int port)

  *Base constructor for any objects implementing the **ProtocolAnalysis** (*p. *380) abstract class. Ensures the base class constructor (**Agent** (*p. *298)) is called an* *also sets the agent type to "PA".*

Parameters

host	*The host name of the AMQP server to connect to.*
port	*The port number of the AMQP server to connect to.*

- override void **init** ()

  This method calls the base class init, and then initializes both the source and request channels to facilitate communications with other agents.

- override void **onTerminate** ()

  Closes the source and request channel before invoking the base class on terminate routines.

- abstract void **processPacket** (PosixTimeval time, Packet nextpacket)

  Protocol Analysis Agents must implement a process packet method to handle incoming packets.

Protected Attributes

- IModel **sourceChannel_**

  A channel for receiving packets.

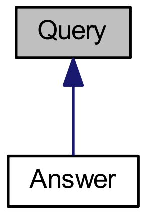
- IModel **requestChannel_**

  A channel for receiving requests.

A.7.2 Query Class Reference

Class to store query information from a DNS request or response.

Inheritance diagram for Query:



Public Member Functions

- virtual void **print** ()

  Prints the relevant values for this DNS query.

Public Attributes

- string **name_**

  The host name contained by the DNS query.

- byte[ ] **rawName_**

  The raw bytes that make up the DNS query name.

- ushort **type_**

  The resource record type that should be returned for this query. For example: A, MX, NS, PTR, etc.

- ushort **class_**

  The class of the associated query.

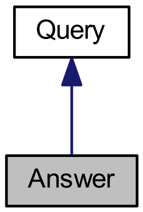
- ushort **errorCode_**

  An error code if there is an error while processing the query.

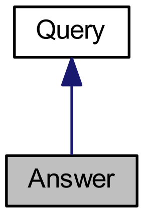
A.7.3 Answer Class Reference

Class that extends the **Query** (p. 382) to add additional member storage for DNS answer records. Note that the IP and data member fields are mutually exclusive. Only one should contain a value depending on the answer type.

Inheritance diagram for Answer:



Collaboration diagram for Answer:



Public Member Functions

- override void **print** ()

  Given an answer record, print out the relevant values.

Public Attributes

- uint **ttl_**

  The time-to-live for this record, used for DNS caching.

- ushort **dataLen_**

  Length of the of the rdata contained in this record.

- IPAddress **ip_**

  The IP address corresponding to the query, if there is one.

- string **data_**

  The alphanumeric data corresponding to the query, if there is any.

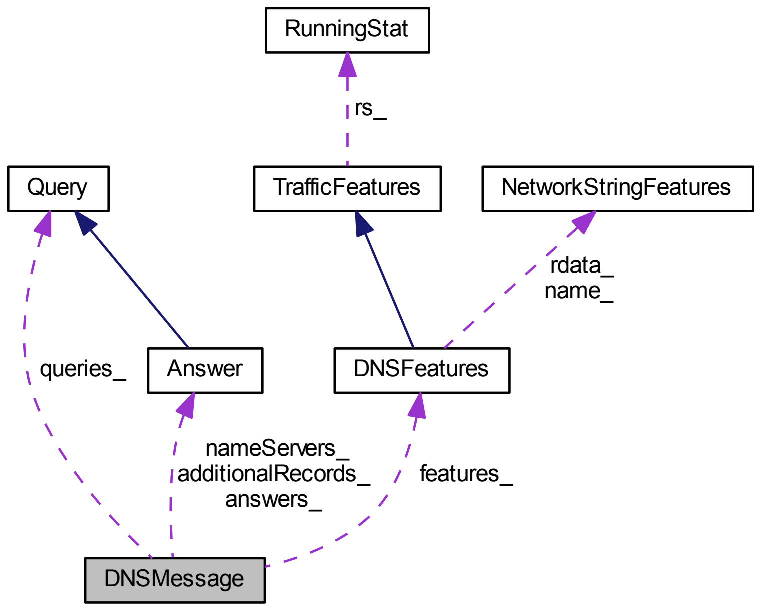
- byte[] **rawRdata_**

  The raw bytes retrieved from the data section of the DNS record.

A.7.4 DNSMessage Class Reference

Represents a single DNS message and all of the records contained within it.

Collaboration diagram for DNSMessage:



Public Member Functions

- **DNSMessage** (UdpPacket udppacket)

  Given a PacketDotNet UdpPacket create a DNS message. Parse out the individual records and store them in lists of ***Query*** *(*p. *382) and* ***Answer*** *(*p. *384) objects.*

- void **parseQuery** (**Query** query)

  Given a query object, parse this object payload for a dns query starting from the currOffset. When this method is called this objects currOffset should be pointed at the len byte for the next query records.

- void **parseRecord** (**Answer** record)

Parse a record from the currOffset position of this objects payload. Note that the record should begin with a 4 byte ttl value. The record could be one of a number of different types including an answer record, a name server or an additional record.

- void **parsePayload** ()

  Parse a complete DNS Message payload. The constructor for this object will have already parsed the DNS header. This method uses the information from the DNS header to figure out how many Queries, Answers, Name Servers and Additional Records there are, parsing each in turn.

- byte[ ] **parseRawName** ()

  Method to parse out the raw bytes representing the query hostname.

- string **parseHostname** ()

  This method can be called to parse out hostnames from a DNS message starting at the currOffset. The method assumes that the len for the hostname is at the currOffset and will handle cases where the len is greater then 192 (indicating a back reference to a previous point in the udp payload).

- string **parseBackReference** (int offset)

  Parse a back reference using a temporary offset that will not effect this objects currOffset value. Very similar to the parseHostName method. Assumes that the offset points at the len of the hostname to extract.

- void **printHeader** ()

  Print the DNS Header.

- void **printRecords** ()

  Print all the records stored in this ***DNSMessage*** *(*p. *385).*

Public Attributes

- **DNSFeatures features_**

  The features associated with this DNS message.

- UInt16 **identifier_**

  Identifier created by the program to represent the query, used to match requests to replies.

- UInt16 **flags_**

  The flag bits including the bits aa, tc, rd, ra, and z.

- byte **qr_**

  Value of the request/response flag extracted from the DNS header.

- UInt16 **opCode_**

  Operation code, indicates the service operation for this DNS message.

- byte **aa_**

  Flag indicating whether or not this is an authoritative answer.

- byte **tc_**

  Flag indicating whether or not truncation is enabled.

- byte **rd_**

  Flag indicating whether or not recursion is desired.

- byte **ra_**

  Flag indicating whether or not recursion is available.

- byte **z_**

Reserved bits that should be set to 0.

- UInt16 **rCode_**

  The return code set to 0 for successful responses and an error code otherwise.

- UInt16 **qCount_**

  The number of questions in the DNS message.

- UInt16 **aCount_**

  The number of answers in the DNS message.

- UInt16 **nsCount_**

  The number of name servers in the DNS message.

- UInt16 **arCount_**

  The number of additional records in the DNS message.

- **Query**[ ] **queries_**

  A list of query records parsed from this DNS message.

- **Answer**[ ] **answers_**

  A list of answer records parsed from this DNS message.

- **Answer**[ ] **nameServers_**

  A list of name servers parsed from this DNS message.

- **Answer**[ ] **additionalRecords_**

  A list of additional resource records parsed from this DNS message.

- const int **DNSHEADERLEN** = 12

  Constant value used for skipping past the DNS header.

Private Attributes

- byte[ ] **littleEndianBytes** = new byte[4]

  A place holder for converting network byte order values (big endian) to host byte order values (little endian)

- int **currOffset**

  The current offset in the DNS message, used while parsing the records.

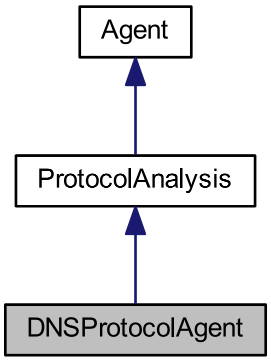
- byte[ ] **payload**

  The complete DNS packet payload.

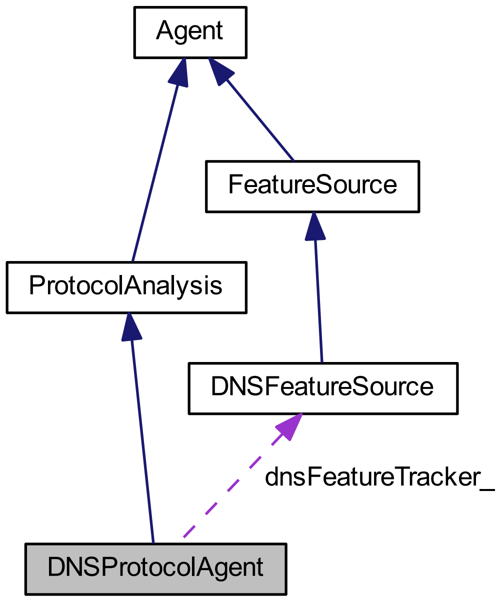
A.7.5 DNSProtocolAgent Class Reference

DNS Resolver Agent is an instance of a Protocol Analysis Agent. It gathers information about DNS Messages received from Traffic Source Agents on the network and provides other agents with the capability to analyze trends in DNS requests and responses.

Inheritance diagram for DNSProtocolAgent:



Collaboration diagram for DNSProtocolAgent:



Public Member Functions

- **DNSProtocolAgent** (string host, int port)

  Constructor initializes all the agent's members, and creates a resident DNSFeature-Source.

- override void **init** ()

  Subscribes to a traffic source agent. The DNS Resolver agent will subscribe to all TCP and UDP traffic with a source or destination port of 53, the standard DNS port. Additionally, the agent will set up a DNS request queue to accept requests from other agents for DNS information.

- override void **onReset** ()

  Clear out information cached about hosts and IP addresses the agent has observed in traffic so far, then call the parent class on reset routines.

- override bool **sensePlanAct** ()

  The sense, plan, act cycle receives DNS messages from a traffic source, extracts DNS messages from the DNS Message Queue, then processes each message.

- void **extractARecords** (**DNSMessage** dns)

  Find all the A records contained in a specific DNS Message and store them in the IP/DNS Record dictionary.

- **Answer**[] **processIPtoHost** (IPAddress requestIp)

  Accept an IP address from an Agent and return a list of hostnames (or the most recent hostname) from the records stored in the dictionary.

- **Answer**[] **processHosttoIP** (string host)

  Accept a host name from an Agent and return a list of answer records from the records stored in the dictionary.

- override void **processPacket** (PosixTimeval time, PacketDotNet.Packet nextpacket)

  Read a message from a Traffic Source Agent. Validate that the packet is a DNS Message and insert it into the Queue of DNS Messages waiting to be processed by the sense, plan, act cycle.

Protected Attributes

- QueuingBasicConsumer **sourceConsumer_**

Basic Consumer to consume messages from the traffic source.

- QueuingBasicConsumer **requestConsumer_**

  Basic Consumer to consume requests from other agents regarding IP addresses.

Private Member Functions

- byte[] **ipsToHosts** (byte[] requestBody)

  Parse out a list of host names from a DNS Host request and return a response containing the IP addresses that the host names have resolved to recently.

- byte[] **hostsToIps** (byte[] requestBody)

  Parse out a list of host names from a DNS Host request and return a response containing the IP addresses that the host names have resolved to recently.

- byte[] **unknownRequest** ()

  Handle instances where we received an AMQPDNSMessageType that we do not recognize. Return a message containing a AMQPDNSMessageType of Unknown-Request.

Static Private Member Functions

- static void **Main** (string[] args)

  Create an instance of a DNS Resolver Agent.

Private Attributes

- byte[] **ipbuffer_**

  1500 bytes buffer to temporarily store incoming packets.

- Dictionary< IPAddress, **Answer**[ ]> **ipDictionary**

  A queue to store incoming messages until they are processed.

- Dictionary< string, **Answer**[ ]> **hostDictionary**

  Structure tp keep track of responses by the host Addresses.

- **DNSFeatureSource dnsFeatureTracker_**

  Keep track of DNS features.

A.7.6 DNSResolverClient Class Reference

The DNS Resolver Client accepts lists of either IP addresses or hostnames and then makes requests out to DNS Agents for either a list of IP addresses that each requested host name has resolved to recently or a list of Host names that each requested IP addresses has been associated with recently.

Public Member Functions

- **DNSResolverClient** (string host, int port)

  Constructor for the DNS Resolver client. It instantiates the connection to the - AMQP server and sets up both the request Queue and the call back queue for responses.

- void **sendIPtoHostLookup** (IPAddress[ ] ips)

  Create a request message to send to DNS agents composed of a list of IP addresses. The request will be stored locally until a response with the corresponding correlation id is received.

- void **sendHosttoIPLookup** (string[ ] hosts)

Create a request message to send to DNS agents composed of a list of host names. The request will be stored locally until a response with the corresponding correlation id is received.

- bool **processIPResponse** (BasicDeliverEventArgs ea)

  Given an AMQP message containing the results of a IP address lookup, this method will parse out the host names that have resolved to the IP addresses sent in the request.

- bool **processHostResponse** (BasicDeliverEventArgs ea)

  Given an AMQP message containing the results of a Host name lookup, this method will parse out the IP addresses that this host names have resolved to sent in the request.

- void **processResponse** ()

  Attempt to get a response from the callback queue. Depending on the type of response received perform the appropriate processing.

Static Private Member Functions

- static void **Main** (string[] args)

  Create an instance of a ***DNSResolverClient*** *(*p. *395).*

Private Attributes

- ConnectionFactory **connectionFactory_**

  Connection factory to support AMQP messaging between agents.

- IConnection **connection_**

A connection to an AMQP server.

- IModel **requestChannel_**

  A channel for receiving packets.

- QueuingBasicConsumer **responseConsumer_**

  A consumer for consuming message from the callback queue.

- Dictionary< string, object > **requests**

  A dictionary structure that store pending requests using correlation ids.

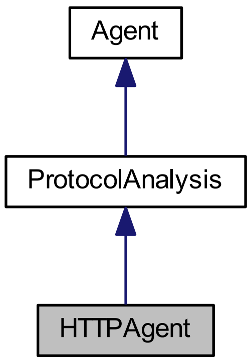
- string **callbackQueue**

  The name of the callback queue to receive DNS responses.

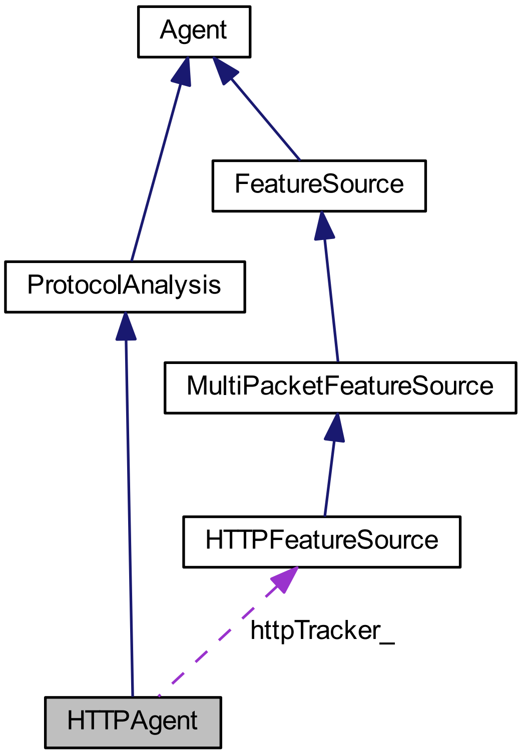
A.7.7 HTTPAgent Class Reference

HTTP Agent is an instance of a Protocol Analysis Agent. It gathers information about HTTP Messages received from Traffic Source Agents on the network and provides other agents with the capability to analyze trends in HTTP requests and responses.

Inheritance diagram for HTTPAgent:



Collaboration diagram for HTTPAgent:



Public Member Functions

- **HTTPAgent** (string host, int port)

  Constructor initializes all the agent's members, and attempts to subscribe to a traffic source agent. The DNS Resolver agent will subscribe to all TCP traffic with a source or destination port of 80 and 8080, standard HTTP ports.

Parameters

host	*The host name of the AMQP server to connect to.*
port	*The port number of the AMQP server to connect to.*

- override BasicDeliverEventArgs **checkControlChannel** ()

  Instruct the resident HTTPFeatureSource to check its control channel before calling the parent classes check control channel.

- override void **init** ()

  Initialize the HTTP Agent by subscribing to a traffic source topic exchange for packets to or from port 80 and 8080 with a protocol values of 6 (TCP).

- override void **onTerminate** ()

  Ensure to invoke the resident HTTPFeatureSource on terminate routines before calling the parent class on terminate routines.

- override bool **sensePlanAct** ()

  Check for new HTTP packets and pass them up to the resident HTTPFeature-Source. Also ensure that any closed HTTP session are published.

- override void **processPacket** (PosixTimeval time, Packet nextpacket)

  Accept an HTTP packet and pass it up to any resident Feature Source agents.

Protected Attributes

- QueuingBasicConsumer **sourceConsumer_**

  Basic Consumer to consume messages from the traffic source.

- **HTTPFeatureSource httpTracker_**

A Traffic Flow Feature Tracker to maintain state on packets and flows observed by the agent.

Static Private Member Functions

- static void **Main** (string[ ] args)

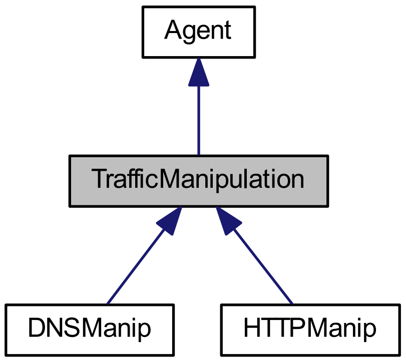
  *Create an instance of an **HTTPAgent** (*p. *411).*

A.8 Traffic Manipulation

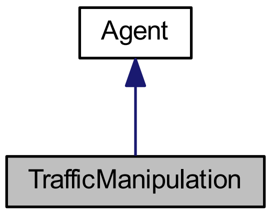
A.8.1 TrafficManipulation Class Reference

A base class providing the API for any agent that will manipulate intercepted packets and send the modified packets.

Inheritance diagram for TrafficManipulation:



Collaboration diagram for TrafficManipulation:



Public Member Functions

- **TrafficManipulation** (string host, int port)

  Construct a Traffic Manipulation Agent The agent will pass the host and port to its parent constructor to set up the connection to the AMQP server.

- override void **init** ()

  Initialize a connection to the label source to receive labels from other agents in the system.

- abstract PacketDotNet.Packet **ModifyPacket** (PacketDotNet.Packet spacket)

  *Each **TrafficManipulation** (p. 401) agent must provide a routine that accepts a packets, modifies the packet and returns a valid packet that will be sent in place of the original packet. This is where the logic goes for deriving the new packet contents.*

- PosixTimeval **sendPacket** (PacketDotNet.Packet packet)

*Send a packet to the intended recipient. This packet will be modified from the original one passed into the **TrafficManipulation** (p. 401) Agent.*

Protected Attributes

- IModel **labelChannel_**

  A channel for communications with machine learning agents.

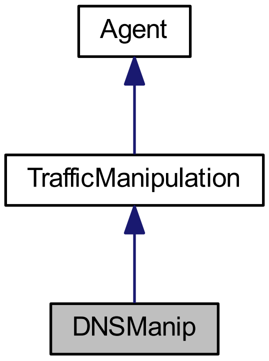
- QueuingBasicConsumer **labelConsumer_**

  Basic Consumer to consume messages from machine learning agents.

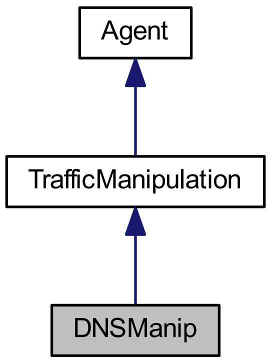
A.8.2 DNSManip Class Reference

Modify DNS request and responses to elicit desired behaviour from malicious software.

Inheritance diagram for DNSManip:



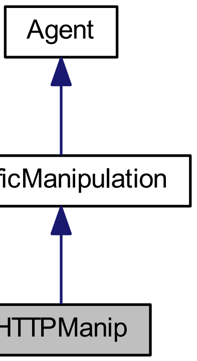
Collaboration diagram for DNSManip:



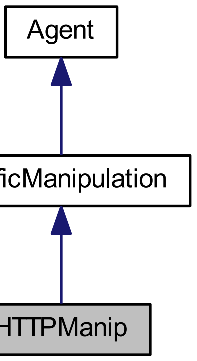
A.8.3 HTTPManip Class Reference

The HTTP Manipulation agent modifies HTTP methods to mimic behaviour, elicit responses from servers.

Inheritance diagram for HTTPManip:



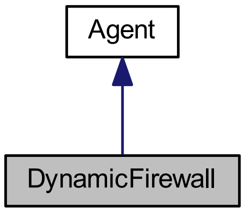
Collaboration diagram for HTTPManip:



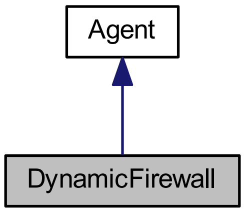
A.8.4 DynamicFirewall Class Reference

The Dynamic Firewall Agent is responsible for managing changes to the firewalls of the various clients in the system to ensure that when traffic is being manipulated, the target the traffic is being manipulated for doesn't attempt further communications with the remote machine.

Inheritance diagram for DynamicFirewall:



Collaboration diagram for DynamicFirewall:



Public Member Functions

- void **sendRule** ()

  Send a new rule to a client machine for inclusion in their current firewall setup.

- void **readRules** ()

  Request a list of the current rules running on a client's firewall.

- void **removeRules** ()

  Remove a specific rule from a client's firewall.

- void **processRuleRequest** ()

  Process an incoming request from another agent to block traffic to a specific client. Blocking is done based on some combination of the five tuple.

Protected Attributes

- IModel **ruleChannel_**

  A channel for communications with machine learning agents.

- QueuingBasicConsumer **ruleConsumer_**

  Basic Consumer to consume messages from machine learning agents.

Private Attributes

- List< String > **firewallRules**

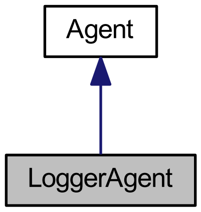
  Current list of firewall rules on the client.

A.9 Observer

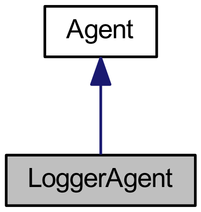
A.9.1 LoggerAgent Class Reference

A Logger agent is passive. It subscribes to various exchanges, collects messages generated by the agents participating in the system and writes them out to a number of different log files.

Inheritance diagram for LoggerAgent:



Collaboration diagram for LoggerAgent:



Public Member Functions

- **LoggerAgent** (string host, int port, string featureFile, string logFile, string labelFile)

  Construct a Logger Agent intended to log features, label and agent logs. The agent will pass the host and port to its parent constructor to set up the connection to the AMQP server.

- override void **init** ()

  Subscribe to the various exchanges required to receive the features, labels and logs. The exchanges are "Features", "Labelled" and "AgentLog" respectively. Also attempts to open the feature file, the label file and the log file for writing. Finally, it starts the stop watch for generating millisecond log time stamps.

- override bool **sensePlanAct** ()

  The sense plan act cycle consists of checking all incoming messages from each subscribed source and writing them out to the log files.

- override void **onTerminate** ()

  Ensure all the text writers flush their buffers before calling the parent classes on terminate routines.

Protected Attributes

- IModel **logChannel_**

  A channel for subscription to logs from other agents.

- QueueingBasicConsumer **logConsumer_**

  Basic Consumer to consume logs from all agents.

- IModel **featureChannel_**

  A channel for communications with feature source agents.

- QueueingBasicConsumer **featureConsumer_**

  Basic Consumer to consume messages from feature source agents.

- IModel **labelChannel_**

  A channel for communications with machine learning agents.

- QueueingBasicConsumer **labelConsumer_**

  Basic Consumer to consume messages from machine learning agents.

Private Member Functions

- void **writeFeature** ()

  Checks the Feature exchange for queue'd features from one or more traffic sources, deserializes the feature and writes it to disk as a UTF8 string.

- void **writeLog** ()

  Checks the Agent Log exchange for queue'd log messages from any agent with logging enabled and writes it to disk.

- void **writeLabel** ()

  Checks the Labelled exchange for queue'd labelled features from one or more machine learning agents, deserializes the feature and writes it to disk as a UTF8 string.

Static Private Member Functions

- static void **Main** (string[ ] args)

Create an instance of a Logger Agent.

Private Attributes

- TextWriter **featureTW_**

  Writer to write received features out to a feature log file.

- TextWriter **logTW_**

  Writer to write received logs out to an agent log file.

- TextWriter **labelTW_**

  Writer to write received labelled features out to a label log file.

- string **featureFile_**

  A file name to write features out to.

- string **logFile_**

  A file name to write logs out to.

- string **labelFile_**

  A file name to write labels out to.

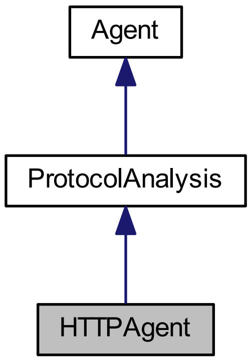
- Stopwatch **st_**

  A stop watch to generate time stamps for the log, feature and label entries.

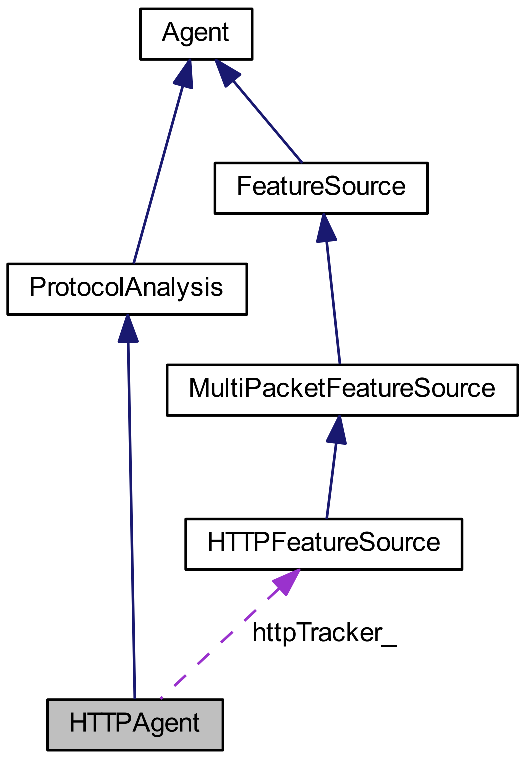
A.9.2 HTTPAgent Class Reference

HTTP Agent is an instance of a Protocol Analysis Agent. It gathers information about HTTP Messages received from Traffic Source Agents on the network and provides other agents with the capability to analyze trends in HTTP requests and responses.

Inheritance diagram for HTTPAgent:



Collaboration diagram for HTTPAgent:



Public Member Functions

- **HTTPAgent** (string host, int port)

  Constructor initializes all the agent's members, and attempts to subscribe to a traffic source agent. The DNS Resolver agent will subscribe to all TCP traffic with a source or destination port of 80 and 8080, standard HTTP ports.

Parameters

host	*The host name of the AMQP server to connect to.*
port	*The port number of the AMQP server to connect to.*

- override BasicDeliverEventArgs **checkControlChannel** ()

  Instruct the resident HTTPFeatureSource to check its control channel before calling the parent classes check control channel.

- override void **init** ()

  Initialize the HTTP Agent by subscribing to a traffic source topic exchange for packets to or from port 80 and 8080 with a protocol values of 6 (TCP).

- override void **onTerminate** ()

  Ensure to invoke the resident HTTPFeatureSource on terminate routines before calling the parent class on terminate routines.

- override bool **sensePlanAct** ()

  Check for new HTTP packets and pass them up to the resident HTTPFeature-Source. Also ensure that any closed HTTP session are published.

- override void **processPacket** (PosixTimeval time, Packet nextpacket)

  Accept an HTTP packet and pass it up to any resident Feature Source agents.

Protected Attributes

- QueuingBasicConsumer **sourceConsumer_**

  Basic Consumer to consume messages from the traffic source.

- **HTTPFeatureSource httpTracker_**

A Traffic Flow Feature Tracker to maintain state on packets and flows observed by the agent.

Static Private Member Functions

- static void **Main** (string[ ] args)

  Create an instance of an ***HTTPAgent*** *(*p. *411).*

A.9.3 MainWindow Class Reference

The ExperimentInterface gives an analyst a way to set up a series of experiments and manages launching various agents.

Public Member Functions

- **MainWindow** ()

  Initialization for the main window.

Protected Attributes

- IConnection **connection_**

  A connection to an AMQP server.

- string **AMQPHost_**

  The IP address or host name of the AMQP server.

- int **AMQPPort_**

  The port number that the AMQP server is listening on.

Private Member Functions

- void **Auto_Start** (object sender, ElapsedEventArgs e)

  Broadcasts the start message to the control channel on an automated timer when moving from the end of one trial to the start of the next one.

- void **Click_Start** (object sender, RoutedEventArgs e)

  Wired to the "Start" button. Broadcasts the start message to the control channel when the user clicks on the button.

- void **Click_Reset** (object sender, RoutedEventArgs e)

  Wired to the "Reset" button. Broadcasts the reset message to the control channel when the user clicks on the button.

- void **Click_Term** (object sender, RoutedEventArgs e)

  Wired to the "Term" button. Broadcasts the terminate message to the control channel when the user clicks on the button.

- void **Click_Pause** (object sender, RoutedEventArgs e)

  Wired to the "Pause" button. Broadcasts the pause message to the control channel when the user clicks on the button.

- void **Click_Add** (object sender, RoutedEventArgs e)

  Wired to the "Add" button. Opens a dialog box allowing the user to add configurations to the list of trials to be performed.

- void **Click_Remove** (object sender, RoutedEventArgs e)

  Wired to the "Remove" button. Allows the user to remove the selected trial from the list of trials to be performed.

- void **Click_Connect** (object sender, RoutedEventArgs e)

  Wired to the "Connect" button. Connects the the AMQP server and sets up the first trial.

- void **startTrial** ()

  Start a trial by connecting up to the AMQP server, launch all local agents and broadcast the required messages to start remote agents. Start a timer to give agents time to start up before broadcasting the "Start" message to the agent control channel.

- void **listenForAgents** ()

  Set up and launch that worker thread that will monitor the agents involved in the trial.

- void **DoWork** (object sender, DoWorkEventArgs e)

  This method is run on a worker thread to monitor the state of all of the agents throughout the trial and periodically update the user interface. Once a terminate message is received on the control channel the experimenter monitors the control channel for each agent to announce that it has completed processing.

- void **WorkerCompleted** (object sender, RunWorkerCompletedEventArgs e)

  When the trial is completed the experimenter continues to poll agents to ensure that all agents shut down cleanly. Then this method will reset the user interface in preparation for the next trial and invoke the startTrial method.

- void **checkControlChannel** ()

  Check the agent control channel for control message broadcast by other agents, such as "Announce", "Terminate" or "Complete".

- void **checkLogChannel** ()

  Check for messages from the agent log exchange. Process any message received to update the experimenter user interface.

- bool **checkAgents** ()

  Check if all the agents have finished processing properly and announced a completed state.

- bool **checkProcesses** ()

  Check if the agent processed managed by this experimenter have exited.

Private Attributes

- ConnectionFactory **connectionFactory_**

  Connection factory to support AMQP messaging between agents.

- IModel **agentControlChannel_**

  A channel for broadcasting and receiving messages to all agents.

- QueueingBasicConsumer **agentControlConsumer_**

  A consumer for control messages broadcast to all agents.

- QueueingBasicConsumer **agentLogConsumer_**

  A consumer for loading messages broadcast by participating agents.

- LinkedList< Process > **agentProcesses_**

  A list of local processes managed by this experimenter.

- BackgroundWorker **worker_**

  A background working thread to manage agents so that the UI doesn't lag.

- LinkedList< string > **agentIds_**

  A list of the unique ids of all agent involved in a trial.

- LinkedList< string > **agentState_**

  A list of current states of agent involved in a trial.

- LinkedList< TextBox > **agentBoxes_**

  A textbox for each agent in the trial displaying their unique id.

- System.Text.Encoding **enc_**

  An encoding to convert arrays of bytes into UTF8 strings.

- Dictionary< string, Brush > **stateColors_**

  Colors available to identify agent states.

- int **currentTrial_**

  The current executing trial.

- long **totalPackets_**

  Total number of packets expected in a trial.

- long **packetsProcessed_**

  Total number of packets processed in a trial.

- long **totalAlerts_**

  Total number of alerts expected in a trial.

- long **alertsProcessed_**

  Total number of alerts published by alert sources in a trial.

- long **featuresPublished_**

Total number of features published during a trial.

- long **featuresLabelled_**

  Total number of features labelled during a trial.

- double **pps_**

  The packets per second achieved by Traffic Sources Agents associated with a trial.

- double **mbps_**

  The megabits per second achieved by Traffic Source Agents associated with a trial.

- double **trainError_**

  The training error for Machine Learning Agents associated with a trial.

- double **testError_**

  The testing error for Machine Learning Agents associated with a trial.

- bool **finished_** = false

  Indicates when all agents have completed their work and ready for the next trial.

Static Private Attributes

- static System.Timers.Timer **startTimer_**

  A timer to manage lag between the end of one trial and the start of the next to give agents a change to shutdown.

Appendix B

Malicious Data Set Rule Hits

The following tables illustrate the number of alerts the Alert Source Agents assigned to feature sets for the Malcious Data Set experiment. There is one table for each traffic type, TCP, UDP, DNS, and HTTP. The table shows the number of alerts generates (column ASA) and the Snort signature ID for the rule that fired against the traffic (column SigID).

ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID
1	10483	1	12609	2	2446	5	648	1	2092
1	15302	1	1941	1	2088	8003	1952	29417	579
2	2256	1	1277	1	1394	1	12626	1	17544

Table B.1: Summary of Alert Source Agent (ASA) labels on UDP traffic (excluding DNS).

ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID
1482644	254	1	15991						

Table B.2: Summary of Alert Source Agent (ASA) labels on DNS traffic.

ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID
1	16155	3	19444	1	15707	1	7864	1	16186
1	6684	1	6681	1	18304	1	18193	1	6509
1	6686	1	7942	1	18307	12	20130	1	19124
15	17410	1	16543	1	15104	16	648	1	19956
2	17648	1	16366	12	7896	1	17260	1	13823
1	17442	1	20651	1	15477	1	18931	1	20656
1	15362	2	13822	2	20641	1	20648	1	13971
1	20732	2	20728	1	20731	38	13990	1	19439
1	15687	1	15460	5	18682	1	15357	1	15462
1	9129	1	16586	1	17548	1	16392	1	4145
1	17103	1	18612	9	1002	1	15461	1	3148
1	16548	1	15861	1	8416	1	18299	1	17588
2	20820	1	8058	2	19177	1	20650	1	20654
1	20652	1	20657	1	19621	1	20663	1	20680
1	18333	3	15258	3	18685	1	15082	1	15107
3	15504	1	15506	1	15505	1	15499	1	16157
2	15469	1	15693	1	15517	1	5711	1	17344
1	15105	1	16560	1	15913	1	15854	4	18495
1	20647	1	20643	1	17075	8	19074	1	16591
1	16037	1	17654	3	17400	1	18301	1	17222
1	18264	1	16044	1	15305	1	15126	1	15116
1	15122	1	15109	1	15114	1	15157	4	16151
1	15540	1	15531	1	14656	12	17322	3	15306
1	17216	1	20137	1	16367	1	16231	1	15090
1	15098	1	19894	1	16412	1	14657	1	14645
2	17572	1	15094	1	16359	1	14262	1	8066
1	14643	1	2435	5	12280	17	1394	1	10162
1	7502	1	16426	1	15084	10	10214	1	13960
1	13828	1	13963	1	13980	1	16506	1	20584
1	13961	1	14255	1	18583	1	18613	2	13573
1	13474	1	3079	1	17232	1	17231	1	9625
5	8375	1	17220	1	12269	4	19174	1	7928
1	17421	1	7944	1	7958	1	7934	1	7938
1	15703	1	17385	1	13830	1	18178	2	15147
1	18296	1	17323	47	10504	1	12448	1	16482
1	18925	1	16555	3	3813				

Table B.3: Summary of Alert Source Agent (ASA) labels on HTTP Traffic.

ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID	ASA	SigID
1	19444	2	2338	3	17410	35407	2181	17	648
2	4918	1	18472	1	2087	1	2253	1	11442
1	11945	1	1390	1	15527	2	16444	1	18509
1	10208	1	14725	6	14737	1	12977	1	2374
1	16524	1	18751	1	15264	32100	1729	2	15255
1	3697	1	18555	1	11289	3	560	1	3218
8	19274	2	19280	1	604	1	18682	2	2044
2	17668	1	2666	1	606	1	20602	1	20601
5	12800	1	15256	1	12069	1	2649	1	15504
2	17344	1	16709	1	542	1	19551	1	12082
1	12597	1	18807	1	18317	4	14782	4	7209
1	9132	1	9027	3	6584	1	3967	1	10018
2	3409	1	10024	1	20662	1	17322	1	16595
2	1866	2	20130	2	16543	208	1394	1	12489
1	13367	2	15213	1	15221	1	15206	2	15199
2	15930	1	7035	1	20440	1	20670	4	2508
1	2936	1	2101	1	8925	1	18512	1	16417
10	12802	3	12798	1	19291	1	10997		

Table B.4: Summary of Alert Source Agent (ASA) labels on TCP traffic (excluding HTTP).

Bibliography

A. Abraham, R. Jain, J. Thomas, and S. Y. Han. D-SCIDS: Distributed soft computing intrusion detection system. *Journal of Network and Computer Applications*, 30:81–98, 2007.

P. Agre and D. Chapman. What are plans for? In P. Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 17–34. MIT Press, Mar. 1991.

A. Al-Rawi and A. Lansari. Effect of Windows XP firewall on network simulation and testing. *Issues in Informing Science and Information Technology*, 4, 2007.

I. Aleksander and H. Morton. *An Introduction to Neural Computing.* International Thomson Computer Press, 1995. ISBN 9781850321675.

M. Alhabeeb, A. Almuhaideb, P. D. Le, and B. Srinivasan. Information security threats classification pyramid. In *AINA Workshops*, pages 208–213. IEEE Computer Soceity, 2010.

A. Ali, A. Saleh, and T. Ramdan. Multilayer perceptrons networks for an intelligent adaptive intrusion detection system. *International Journal of Computer Science and Network Security*, 10(2), Feb. 2010.

J. Allen. Imitation learning from multiple demonstrators using global vision. Master's thesis, Department of Computer Science, University of Manitoba, Winnipeg, Canada, Aug. 2009.

E. Alpaydin. *Introduction To Machine Learning*. Adaptive Computation and Machine Learning. Mit Press, 2004. ISBN 9780262012119.

T. Alpean, C. Bauckhage, and A.-D. Schmidt. A probabilistic diffusion scheme for anomaly detection on smartphones. In *Workshop in Information Security Theory and Practices*, pages 31–46, 2010.

Álvaro Herrero, E. Corchado, M. Pellicer, and A. Abraham. MOVIH-IDS: A mobile-visualization hybrid intrusion detection system. *Neurocomputing*, 72(13-15):2775–2784, 2009.

K. Anagnostakis, M. Greenwald, S. Ioannidi, and A. Keromytis. COVERAGE: Detecting and reacting to worm epidemics using cooperation and validation. *International Journal of Information Security*, 6(6):361–378, 2007.

J. Anderson, B. Tanner, and R. Wegner. Peer reinforcement in homogeneous and heterogeneous multi-agent learning. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Soft Computing (ASC2002)*, pages 13–18, Banff, AB, July 2002a.

J. Anderson, R. Wegner, and B. Tanner. Exploiting opportunities through dynamic coalitions in robotic soccer. In *Proceedings of the AAAI International Workshop on Coalition Formation in Dynamic Multiagent Environments*, Edmonton, AB, July 2002b.

J. Anderson, J. Baltes, and J. Kraut. The Keystone Rescue Team. In D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors, *The Seventh RoboCup Competitions and Conferences*, Padova, Italy, July 2003.

S. Andersson, A. Clark, and G. Mohay. Network-based buffer overflow detection by exploit code analysis. In *AusCERT Information Technology Security Conference*, 2004.

R. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112.

R. Arkin. Behavior-based robot navigation for extended domains. *Adaptive Behavior*, 1(2):201–225, 1992.

R. Arkin. *Behavior-Based Robotics*. Cambridge: MIT Press, Cambridge, MA, 1998.

R. Arkin and T. Balch. AuRa: Principles and practice in review. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2):175–189, 1997.

P. Bacher, T. Holz, M. Kotter, and G. Wicherski. Know your enemy: Tracking botnets, Oct. 2008. URL `http://www.honeynet.org/papers/bots`.

P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes platform: An efficient approach to collect malware. In *Recent Advances in Intrusion Detection*, pages 165–184. Springer Berlin, 2006.

J. Bagot, J. Anderson, and J. Baltes. Vision-based multi-agent slam for humanoid robots. In *Proceedings of the 5th International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS-2008)*, pages 171–176, June 2008.

E. Balas and C. Viecco. Towards a third generation data capture architecture for honeynets. In *6th IEEE Information Assurance Workshop*. IEEE, 2005.

T. Balch and R. Arkin. Motor schema-based formation control for multiagent robot teams. In *Proceedings of the 1995 International Conference on Multiagent Systems*, pages 10–16, San Francisco, CA, 1995.

J. Baltes and J. Anderson. Intelligent global vision for teams of mobile robots. In S. Kolski, editor, *Mobile Robots: Perception and Navigation*, chapter 9, pages 165–186. Advanced Robotic Systems International, Vienna, Austria, 2007.

J. Baltes, M. Mayer, J. Anderson, K.-Y. Tu, and A. Liu. The humanoid leagues in robot soccer competitions. In *Proceedings of the IJCAI Workshop on Competitions in Artificial Intelligence and Robotics*, pages 9–16, Pasadena, California, July 2009. AAAI Press.

S. Bandini, S. Manzoni, and G. Vizzari. Multi-agent approach to localization problems: Multilayered multi-agent situated system. *Web Intelligence and Agent Systems*, 2(3):155–166, Oct. 2004.

B. Banerjee, L. Kraemer, and J. Lyle. Multi-agent plan recognition: Formalization and algorithms. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

Y. Beyene, M. Faloutsos, and H. Madhyastha. SyFi: A systematic approach for estimating stateful firewall performance. In N. Taft and F. Ricciato, editors, *Passive and Active Measurement 13th International Conference*, volume 7192 of *Lecture Notes in Computer Science*, pages 74–84. Springer, Mar. 2012.

A. Bordes, L. Bottou, P. Gallinari, and J. Weston. Solving multiclass support vector machines with larank. In *In Proceedings of the ICML*, 2007.

I. BreakingPoint Systems. Breakingpoint, May 2012. URL `http://www.breakingpointsystems.com`.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, Mar. 1986.

R. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. In *Proceedings of the International Conference on Robotics and Automation*, pages 692–694, May 1989.

R. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1&2): 3–15, June 1990.

R. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991a.

R. Brooks. New approaches to robotics. *Science*, 253:1227–1232, Sept. 1991b.

R. Brooks, C. Breazeal, M. Marjanovic, B. Scassellati, and M. Williamson. The cog project: Building a humanoid robot. pages 52–87. Springer, New York, 1999.

J. Cameron, D. MacKenzie, K. Ward, R. Arkin, and W. Book. Reactive control for mobile manipulation. In *Proceedings of the International Conference on Robotics and Automation*, pages 228–235, Atlanta, GA, 1993.

S. Canada. *Canada Year Book 2008*. Statistics Canada, Nov. 2009.

I. Chairunnisa, Lukas, and H. D. Widiputra. Clustering base intrusion detection for network profiling using k-means, ECM and k-nearest neighbor algorithms. In *Konferensi Nasional Sistem dan Informatika 2009*, Nov. 2009.

T. Chan, G. Golub, and R. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, pages 242–247, 1983.

B. Chapman and E. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1995. ISBN 1565921240.

D. Chapman. Penguins can make cake. *AI Magazine*, 10(4):45–50, 1989.

S. Chebrolua, A. Abrahama, and J. Thomas. Feature deduction and ensemble design of intrusion detection systems. *Computers & Security*, 24:295–307, 2005.

H. Chen, J. Clark, S. Shaikh, H. Chivers, and P. Nobles. Optimising IDS sensor placement. In *International Conference on Availability, Reliability and Security*, pages 315–320, 2010.

T. Chen. Trends in viruses and worms. *Internet Protocol Journal*, 6:23–33, Sept. 2003.

T. Chen and J.-M. Robert. *The Evolution of Viruses and Worms*, chapter 16. Marcel Dekker, 2005.

Y. Chen, A. Abraham, and B. Yang. Hybrid flexible neural-tree-based intrusion detection systems. *International Journal of Intelligent Systems*, 22:337–352, 2007.

T. Clarke. Fuzzing for software vulnerability discovery. Technical Report RHUL-MA-2009-04, Royal Holloway University of London, Feb. 2009.

S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: Preliminary report. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI)*, pages 129–135, Menlo Park, CA, 2000. AAAI Press.

P. Cortada Bonjoch, G. Sanromà, and P. Garcia López. IDS based on self-organizing maps. Technical report, RedIRIS, 2002.

J. Crandall, Z. Su, F. Wu, and F. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 235–248, New York, NY, USA, 2005. ACM. ISBN 1-59593-226-7. doi: 10.1145/1102120.1102152. URL `http://doi.acm.org/10.1145/1102120.1102152`.

D. Dasgupta, F. Gonzalez, K. Yallapu, J. Gomez, and R. Yarramsettii. CIDS: An agent-based intrusion detection system. *Computer and Security*, 24:387–398, Aug. 2005.

D. Dash, B. Kveton, J. M. Agosta, E. Schooler, J. Chandrashekar, A. Bachrach, and A. Newman. When gossip is good: Distributed probabilistic inference for detection of slow network intrusions. In *AAAI'06: proceedings of the 21st national conference* *on Artificial intelligence*, pages 1115–1122. AAAI Press, 2006. ISBN 978-1-57735-281-5.

M. de Denus, J. Anderson, and J. Baltes. Heuristic formation control in multi-robot systems using local communication and limited identification. In J. Baltes, M. G. Lagoudakis, T. Naruse, and S. Shiry, editors, *Proceedings of RoboCup-2009: Robot Soccer World Cup XIII*, Graz, Austria, July 2009.

J. de Sá. *Pattern Recognition: Concepts, Methods, and Applications*. Springer, 2001. ISBN 9783540422976. URL `http://books.google.co.uk/books?id=O5vwppJQQwIC`.

K. Deeter, K. Singh, S. Wilson, L. Filipozzi, and S. Vuong. APHIDS: A mobile agent-based programmable hybrid intrusion detection system. *Mobility Aware Technologies and Applications*, 3284:244–253, 2004.

R. Deibert and R. Rohozinskli. Tracking GhostNet: Investigating a cyber espionage network. Technical Report JR02-2009, Information Warfare Monitor, Mar. 2009.

R. Deibert and R. Rohozinskli. Shadows in the cloud: Investigating cyber espionage 2.0. Technical Report JR03-2010, Information Warfare Monitor, Apr. 2010.

A. Dewdney. *The Armchair Universe: An Exploration of Computer Worlds*. W. H. Freeman & Co., New York, NY, USA, 1988.

D. Dittrich and S. Dietrich. P2p as botnet command and control: A deeper insight. In *2008 3rd International Conference on Malicious and Unwanted Software*, Oct. 2008.

S. R. Dong, D. Huynh, and N. Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(1):1–25, 2004.

H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *Recent Advances in Intrusion Detection*, pages 135–154. Springer-Verlag Berlin Heidelberg, 2008.

K. Faraoun and A. Boukelif. Neural networks learning improvement using the k-means clustering algorithm to detect network intrusions. *International Journal of Computational Intelligence*, 3(2), 2007.

D. Farid and M. Z. Rahman. Anomaly network intrusion detection based on improved self adaptive bayesian algorithm. *Journal of Computers*, 5(1), Jan. 2010.

D. Farid, J. Darmont, and M. Z. Rahman. Attribute weighting with adaptive NBTree for reducing false positives in intrusion detection. *International Journal of Computer Science and Information Security*, 8(1):19–26, 2010.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`. Updated by RFCs 2817, 5785, 6266, 6585.

A. O. Flaglien. BRO: An intrusion detection system. Manual from application, Nov. 2007.

J. Franklin and V. Paxson. An inquiry into the nature and causes of the wealth of Internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on*

Computer and communications security, pages 375–388, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2.

T. Gal. SharpPcap: A packet catpure framework for .NET, Apr. 2012. URL `http://www.codeproject.com/Articles/12458/SharpPcap-A-Packet-Capture-Framework-for-NET`.

M. Gashler. Waffles: A machine learning toolkit. *Journal of machine learning research*, pages 2383–2387, 2011.

E. Gat. Integrating planning and reaction in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI)*, pages 809–815, San Jose, CA, July 1992.

C. Geib and R. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009. ISSN 0004-3702.

L. Girardin. An eye on network intruder-administrator shootouts. In *Workshop on Intrusion Detection and Network Monitoring*, pages 19–28, 1999.

V. T. Goh, J. Zimmermann, and M. Looi. Towards intrusion detection for encrypted networks. In *4th International Conference on Availability, Reliability and Security*, pages 540–545. IEEE Computer Society, Mar. 2009.

V. T. Goh, J. Zimmermann, and M. Looi. Experimenting with an intrusion detection system for encrypted networks. *International Journal Bus. Intelligence Data*

Mining, 5(2):172–191, 2010. ISSN 1743-8195. doi: http://dx.doi.org/10.1504/IJBIDM.2010.031286.

V. Golovko, S. Bezobrazov, P. Kachurka, and L. Vaitsekhovich. *Neural Network and Artificial Immune Systems for Malware and Network Intrusion Detection*, pages 485–513. Springer Berlin, 2010.

J. A. Gonzalez. Numerical analysis for relevant features in intrusion detection. Master's thesis, Air Force Institute of Technology, Mar. 2009.

D. Goodin. Insulin pump hack delivers fatal dosage over the air, Oct. 2011. URL `http://www.theregister.co.uk/2011/10/27/fatal_insulin_pump_attack`.

N. Gornitz, M. Kloft, K. Rieck, and U. Brefeld. Active learning for network intrusion detection. In *2nd ACM workshop on security and artificial intelligence*, pages 47–54, 2009.

S. Graves and R. Volz. Action selection in teleautonomous systems. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 14–19, 1995.

J. Greensmith, U. Aickelin, and G. Tedesco. Information fusion for anomaly detection with the dendritic cell algorithm. *Information Fusion*, 11(1):21–34, 2010. ISSN 1566-2535.

J. Grizzard, V. Sharma, C. Nunnery, B. B. King, and D. Dagon. Peer-to-peer botnets: Overview and case study. In *USENIX HotBots 2007*, 2007.

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*, Aug. 2007.

G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering analysis of network traffic for protocol and structure independent botnet detection. In *17th USENIX Conference on Security Symposium*, pages 139–154, Berkeley, CA, USA, 2008a. USENIX Association.

G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting botnet command and control channels in network traffic. In *15th Annual Network and Distributed System Security Symposium*, Feb. 2008b.

R. Guttman and P. Maes. *Cooperative vs. Competitive Multi-Agent Negotiations in Retail Electronic Commerce*, volume 1435. Springer Berlin, 1998.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009. ISSN 1931-0145. doi: 10.1145/1656274.1656278. URL `http://doi.acm.org/10.1145/1656274.1656278`.

M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2001. USENIX Association.

S. Hansen and T. Atkins. Automated system monitoring and notification with Swatch. In *In Proceedings of the seventh Systems Administration Conference*, Nov. 1993.

S. Harnad. The symbol grounding problem. *Physica D*, 42:335–346, 1990.

A. Herrero and E. Corhado. *Multiagent Systems for Network Intrusion Detection: A Review*, volume 63. Springer Berling/Heidelberg, 2009.

L. Holdings. L0phtCrack - Windows and Unix password auditing and recovery, Apr. 2010. URL `http://www.l0phtcrack.com/`.

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 *1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9, Berkeley, CA, USA, 2008. USENIX Association.

J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *In Proceedings of the National Acedemy of Science USA*, volume 79, pages 2554–2558, Apr. 1982.

I. Horswill. Polly, a vision-based artificial agent. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI)*, pages 824–829, Washington, DC, July 1993.

J. Hu, X. Yu, D. Qiu, and H.-H. Chen. A simple and efficient hidden Markov model scheme for host- based anomaly intrusion detection. *The Magazine of Global Internetworking.*, 23(1):42–47, 2009. ISSN 0890-8044. doi: http://dx.doi.org/10.1109/MNET.2009.4804323.

F. Hugelshofer, P. Smith, D. Hutchison, and N. Race. OpenLIDS: A lightweight intrusion detection system for wireless mesh networks. In *MobiCom '09: Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 309–320, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-702-8. doi: http://doi.acm.org/10.1145/1614320.1614355.

S. Intille and A. Bobick. A framework for recognizing multi-agent action from visual evidence. In *In AAAI-99*, pages 518–525. AAAI Press, 1999.

L. J. Jensen and A. Bateman. The rise and fall of supervised machine learning techniques. *Bioinformatics*, 27(24):3331–3332, 2011. doi: 10.1093/bioinformatics/btr585. URL `http://bioinformatics.oxfordjournals.org/content/27/24/3331.short`.

X. Jiang and X. Wang. Stealthy malware detection and monitoring through VMM-based out of the box semantic view reconstruction. In *ACM Conference on Computer and Communications Security 2007*, pages 1–27, Oct. 2007.

W. Jimenez, A. Mammar, and A. Cavalli. Software vulnerabilities, prevention and detection methods: A review. In *European Workshop on Security in Model Driven Architecture*, June 2009.

P. Kabiri and A. Ghorbani. Research in intrusion detection and response: A survey. *International Journal of Network Security*, pages 84–102, 2005.

U. Kamath, A. Shehu, and K. De Jong. A two-stage evolutionary approach for effective classification of hypersensitive DNA sequences. *J. Bioinformatics and Computational Biology*, 9(3):399–413, 2011.

B. B. Kang, E. Chan-Tin, C. Lee, J. Tyra, H. J. Kang, C. Nunnery, Z. Wadler, G. Sinclair, N. Hooper, D. Dagon, and Y. Kim. Towards complete node enumeration in a peer-to-peer botnet. In *4th International Symposium on Information, Computer, and Communications Security*, pages 23–34, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-394-5. doi: http://doi.acm.org/10.1145/1533057.1533064.

Kaspersky. Kaspersky::home page, June 2010. URL `http://www.kaspersky.com/`.

H. Kautz and J. Allen. Generalized plan recognition. In T. Kehler, editor, *AAAI*, pages 32–37. Morgan Kaufmann, 1986.

T. Kauzoglu. Determining optimum structure for artificial neural networks. In K. Cardiff, editor, *25th Annual Technical Conference and Exhibition of the Remote Sensing Society*, pages 675–682, Sept. 1999.

W. Ketter, J. Collins, M. Gini, A. Gupta, and P. Schrater. Detecting and forcasting economic regimes in multi-agent automated exchanges. *Decision Support Systems*, 47:307–318, 2009.

K.-C. Khor, C.-Y. Ting, and S.-P. Amnuaisuk. From feature selection to building of bayesian classifiers: A network intrusion detection perspective. *American Journal of Applied Sciences*, 6(11):1949–1960, 2009.

M. Kloft, U. Brefeld, P. Dussel, C. Gehl, and P. Laskov. Automatic feature selection for anomaly detection. In *AISEC 2008*, pages 71–76, Oct. 2008.

T. Kohonen. *Self-Organizing Maps.* Springer Series in Information Sciences. Springer, 2001. ISBN 9783540679219.

C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security '09*, Aug. 2009.

M. Krogseter and C. Thomas. Adaptivity: System-initiated individualism. In R. Oppermann, editor, *Adaptive User Support: Ergonomic Design of Manually and Automatically Adaptable Software*, pages 67–96. Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

M. Krogseter, R. Oppermann, and C. Thomas. A user interface integrating adaptability and adaptivity. In R. Oppermann, editor, *Adaptive User Support: Ergonomic Design of Manually and Automatically Adaptable Software*, pages 97–125. Lawrence Erlbaum Associates, Hillsdale, NJ, 1994.

G. Kuhlmann, W. Knox, and P. Stone. Know thine enemy: A champion RoboCup coach agent. In *Twenty-First National Conference on Artificial Intelligence*, 2006.

V. Kurland. What's new in Firewall Builder 3.0. *Linux Journal*, 2010(189):4, 2010. ISSN 1075-3583.

K. Labib and R. Vemuri. NSOM: A real-time network-based intrusion detection system using self-organizing maps. Technical report, Dept. of Applied Science, University of California, Davis, 2002.

T. Lane and C. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2:150–158, 1998.

T. Lauinger, V. Pankakoski, D. Balzarotti, and E. Kirda. Honeybot, your man in the middle for automated social engineering. In *3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2010.

A. Lawniczak, H. Wu, and B. Di Stefano. Entropy based detection of DDoS attacks in packet switching network models. In *Complex Sciences*, volume 5. Springer Berlin Heidelberg, Feb. 2009.

F. Leder and W. Tilmann. Know your enemy: Containing Conficker. Technical Report rev1, The Honeynet Project, Mar. 2009.

J. Lee, M. Hubar, E. Durfee, and P. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *Proceedings of the Conference on Intelligent Robotics in Field, Factory, and Space*, pages 842–849, Houston, TX, Mar. 1994.

W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *In Proceedings of the 7th USENIX Security Symposium*, 1998.

C. Leistner, A. Saffari, and H. Bischof. MIForests: Multiple-instance learning with randomized trees. In *In Proceedings of the eleventh european conference on computer vision*, Sept. 2010.

P. Lichodzijewski, N. Zincir-Heywood, and M. Heywood. Dynamic intrusion detection using self-organizing maps. In *14th Annual Canadian Information Technology Security Symposium*, 2002.

R. Ling. Comparison of several algorithms for computing sample means and variances. *Journal of the American Statistical Association*, (348):859–866, 1983.

C. Livadas, R. Walsh, D. Lapsley, and T. Strayer. Using machine learning techniques to identify botnet traffic. In *2nd IEEE LCN Workshop on Network Security*, pages 967–974, 2006.

I. Lorenzo-Fonseca, F. Maciá-Pérez, F. J. Mora-Gimeno, R. Lau-Fernández, J. A. Gil-Martínez-Abarca, and D. Marcos-Jorquera. Intrusion detection method using neural networks based on the reduction of characteristics. In J. Cabestany, editor, *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517, pages 1296–1303. Springer Berlin/Heidelberg, 2009.

D. Lyons and A. Hendriks. Planning as incremental adaptation of a reactive system. *Robotics and Autonomous Systems*, 14(4):255–288, 1995.

W. Maisel and T. Tadayoshi. Improving the security and privacy of implantable medical devices. *The New England Journal of Medicine*, 362, Apr. 2010.

M. Mataric. Integration of representation into goal driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992.

M. Mataric. Behavior-based control: Examples from navigation: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, (2-3):323–336, 1997.

Mcafee. Mcafee::home page, June 2010. URL `http://www.mcafee.com/us/`.

M. McCaughey and M. Ayers, editors. *Cyberactivim: Online Activism in Theory and Practice*. Routledge, 2003.

P. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), Nov. 1987a. URL `http://www.ietf.org/rfc/rfc1034.txt`. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987b. URL `http://www.ietf.org/rfc/rfc1035.txt`. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

M. A. Mohamed and M. Mohamed. Development of hybrid-multi-stages intrusion detection systems. *International Journal of Computer Science and Network Security*, 10(3):69–77, Mar. 2010.

J. Molina and M. Cukier. Evaluating attack resiliency for host intrusion detection systems. *Journal of Information Assurance and Security*, 4, 2009.

D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1:33–39, July 2003.

A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A crawler-based study of spyware in the web. In *Network and Distributed System Security Symposium 2006*, 2006.

S. Mukkamala and A. Sung. Feature ranking and selection for intrusion detection systems using support vector machines. Technical report, Institute of Minera y Tecnologa, 2003.

A. Newell and H. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 12:113–126, 1976.

J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2339-0. doi: http://dx.doi.org/10.1109/SP.2005.15.

N. Nilsson. Shakey the robot. Technical Report 323, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1984.

S. A. Onashoga, A. Akinde, and A. S. Sodiya. A strategic review of existing mobile agent-based intrustion detection systems. *Issues in Informing Science and Information Technology*, 6, 2009.

A. Papadogiannakis, M. Polychronakis, and E. Markatos. Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In *2010 European Workshop on System Secutiry*, Apr. 2010.

V. D. Parunak. Go to the ant: Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75(0):69–101, Oct. 1997.

V. Patole, V. Pachghare, and P. Kulkarni. Self organizing maps to build intrusion detection system. *International Journal of Computer Applications*, 1(8):1–4, Feb. 2010.

V. Paxson. Bro: A system for detecting network intruders in real-time. 31:2435–2463, Dec. 1999.

S. Peisert, M. Bishop, and K. Marzullo. What do firewalls protect? an empirical study of firewalls, vulnerabilities, and attacks. Technical Report CSE-2010-8, UC Davis Department of Computer Science, Mar. 2010.

Y. Peng, L. Zhang, M. Chang, and Y. Guan. An effective method for combating malicious scripts clickbots. In *Computer Security - ESORICS 2009*, pages 523–538, Sept. 2009.

C. Pinzon, A. Herrero, J. D. Paz, E. Corchado, and J. Bajo. CBRid4SQL: A CBR intrusion detector for SQL injection attacks. In *5th International Conference on Hybrid Artificial Intelligence Systems*, pages 510–519. Springer Berlin, June 2010.

M. Polla, T. Honkela, and T. Kohonen. Bibliography of self-organizing map (SOM) papers: 2002-2005 addendum. Technical Report TKK-ICS-R24, TKK Reports in Information and Computer Science, Helsinki University of Technology, 2009.

P. Porras, H. Sadi, and V. Yegneswaran. A foray into Confickers logic and rendezvous points. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

L. Portnoy. Intrusion detection with unlabeled data using clustering, 2000.

L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *ACM CCS Workshop on Data Mining Applied to Security*. ACM Press, Nov. 2001.

K. Poulsen. New SubSeven Trojan unleashed, Mar. 2001. URL `www.securityfocus.com/news/171`.

N. Provos. A virtual honeypot framework. In *13th USENIX Security Symposium*, pages 1–14, 2004.

N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFrames point to us. Technical Report provos-2008a, Google, 2008.

M. Ramadas, S. Ostermann, and B. Tjaden. Detecting anomalous network traffic with self-organizing maps. In G. Vigna, C. Krgel, and E. Jonsson, editors, *RAID 2003*, volume 2820, pages 36–54. Springer, Heidelberg, Sept. 2003.

M. Rehak, M. Pechoucek, P. Celeda, V. Krmicek, M. Grill, and K. Bartos. Multi-agent approach to network intrusion detection. In *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1695–1696, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

B. C. Rhodes, J. Mahaffey, and J. Cannady. Multiple self-organizing maps for intrusion detection. In *23rd National Information Systems Security Conference*, 2000.

K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov. Botzilla: Detecting the phoning home of malicious software. In *25th Symposium On Applied Computing*, Mar. 2010.

S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach.* Alan Apt, Upper Saddle River, New Jersey, 1995.

A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *In Proceedings of the computer vision workshop 12th international conference on computer vision*, Sept. 2009.

A. Saffari, M. Godec, T. Pock, C. Leistner, and H. Bischof. Online multi-class lpboost. In *In Proceedings of the IEEE computer soceity conference on computer vision and pattern recognition*, June 2010.

Sax2. Sax2::home page, Apr. 2010. URL `http://www.ids-sax2.com/`.

K. Scarfone and P. Mell. Guide to intrusion detection and prevention systems (IDPS). Technical Report 800-94, National Institute of Standards and Technology, Feb. 2007.

M. Sheikham, Z. Jadidi, and M. Beheshti. Effects of feature reduction on the performance of attack recognition by static and dynamic neural networks. *World Applied Sciences Journal*, 8(3):302–308, 2010.

M. Sheikhan and Z. Jadidi. Misuse detection using hybrid of association rule mining and connectionist modeling. *World Applied Sciences Journal*, 7:31–37, 2009.

M. Sheikhan and A. A. Shabani. Fast neural intrusion detection system based on hidden weight optimization algorithm and feature selection. 7:45–53, 2009.

shoki. shoki::home page, Apr. 2010. URL `http://shoki.sourceforge.net`.

Snort. Snort::home page, Apr. 2010. URL `http://www.snort.org/`.

E. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, Dec. 1988.

E. Spafford. Computer virsuses, a form of artificial life? Technical Report CSD-TR-985, Software Engineering Research Center, Purdue University, June 1990.

E. Spafford and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 34(4):547–570, 2000.

springsource. RabbitMQ messaging that just works, Apr. 2012. URL `http://www.rabbitmq.com`.

A. Stamos. Aurora response recommendations. Technical Report 1.0, iSec Partners, Feb. 2010.

U. Steinhoff, A. Wiesmaier, R. Arajo, and M. Lippert. *The State of the Art in DNS Spoofing*, pages 174–189. 2006.

B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *16th ACM Conference on Computer and Communications Security*, Nov. 2009.

L. Strachan, J. Anderson, M. Sneesby, and M. Evans. Minimalist user modelling in a complex commercial software system. *User Modelling and User-Adapted Interaction*, 10:109–145, 2000.

S. Sun and Y. Wang. Research and application of an improved support vector clustering algorithm on anomaly detection. *Journal of Software*, 5(3), Mar. 2010.

A. Sung and S. Mukkamala. Identifying important features for intrusion detection using support vector machines and neural networks. In *International Symposium on Applications and the Internet*, Washington, DC, USA, 2003. IEEE Computer Society.

L. Surfer. Log surfer::home page, Apr. 2010. URL `http://www.crypt.gen.nz/logsurfer`.

Swatch. Swatch::home page, June 2010. URL `http://sourceforge.net/projects/swatch`.

Symantec. Symantec::home page, June 2010. URL `http://www.symantec.com/`.

J. Tarala. Network security: Theory versus practice. Technical report, SANS Institute, May 2011.

S. Teng, H. Du, N. Wu, W. Zhang, and J. Su. A cooperative network intrusion detection based on fuzzy SVMs. *Journal of Networks*, 5(4):475–483, Apr. 2010.

S. Tools. Sentrytools::home page, Apr. 2010. URL `http://sourceforge.net/projects/sentrytools/`.

Z. Trabelsi and W. El-Hajj. On investigating ARP spoofing security solutions. *International Journal Internet Protocol Technology*, 5(1):92–100, 2010.

J. Twycross. *Integrated Innate and Adaptive Artificial Immune Systems Applied to Process Anomaly Detection*. PhD thesis, University of Nottingham, Jan. 2007.

J. Twycross and U. Aickelin. Libtissue - implementing innate immunity. In *IEEE World Congress on Computational Intelligence*, pages 499–506, July 2006.

M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In *Recent Advances in Intrusion Detection 2007*, 2007.

L. Vatisekhovich. Intrusion detection in TCP/IP networks using immune systems paradigm and neural network detectors. In *XI International PhD Workshop*, Oct. 2009.

M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *ACM Symposium on Operating System Principles*, 2005.

J. Wang, X. Hong, R. rong Ren, and T. huang Li. A real-time intrusion detection system based on PSO-SVM. In *2009 International Workshop on Information Security and Application*. Academy Publisher, Nov. 2009.

P. Wang, L. Wu, R. Cunningham, and C. Zou. Honeypot detection in advanced botnet attacks. *International Journal of Information and Computer Security*, 4: 30–51, 2010.

Y.-M. Wang, C. Verbowski, J. Dunagen, Y. Chen, H. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, statebased approach to change and configuration management and support. In *17th USENIX Conference on System Administration*, pages 159–172, Berkeley, CA, USA, Oct. 2003. USENIX Association.

Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. Technical report, Microsoft Research, Redmond, 2006.

R. Wegner. Balancing robotic teleoperation and autonomy in a complex and dynamic environment. Master's thesis, Department of Computer Science, University of Manitoba, July 2003.

R. Wegner and J. Anderson. Agent-based support for balancing teleoperation and autonomy in urban search and rescue. *International Journal of Robotics and Automation*, 21(2), Feb. 2006.

D. Whyte, P. van Oorschot, and E. Kranakis. Exposure maps: Removing reliance on attribution during scan detection. In *USENIX HotSec 2006, 1st Workshop on Hot Topics in Security*, July 2006.

N. Wiebe and J. Anderson. Local methods for supporting grounded communication in robot teams. In D. Liu, L. Wang, and K. C. Tan, editors, *Design and Control of Intelligent Robotic Systems*, chapter 14, pages 279–301. Springer-Verlag, Heidelberg, 2009.

C. Wright, C. Connelly, T. Braje, J. Rabek, L. Rossey, and R. Cunningham. Generating client workloads and high-fidelity network traffic for controllable, repeatable experiments in computer security. In S. Jha, R. Sommer, and C. Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 218–237. Springer Berlin / Heidelberg, 2010.

P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. SWAP: Mitigating XSS attacks using a reverse proxy. In *IWSESS '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3725-2. doi: http://dx.doi.org/10.1109/IWSESS.2009.5068456.

B. Yan, B. Fang, B. Li, and Y. Wang. Detection and defence of DNS spoofing attack. *Jisuanji GongchengComputer Engineering*, 32(21):130–132, 2006.

Y. Yang, D. Jiang, and M. Xia. Using improved GHSOM for intrusion detection. *Journal of Information Assurance and Security*, 5:232–239, 2010.

D. Y. Yeung and Y. Ding. Host based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, 36(1):229–243, 2003.

L. Ying-xu and L. Zeng-hui. *Unkown Malicious Identification*, chapter 26. Springer Science and Business Media, 2009.

J. Yuan, H. Li, S. Ding, and L. Cao. Intrusion detection model based on improved support vector machine. In *Third International Symposium on Intelligent Information Technology and Security Informatics*, pages 465–469, 2010.

S. Zanero and S. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *ACM Symposium on Applied Computing SAC*, pages 41–49, Mar. 2004a.

S. Zanero and S. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied comput-*

ing, pages 412–419, New York, NY, USA, 2004b. ACM. ISBN 1-58113-812-1. doi: http://doi.acm.org/10.1145/967900.967988.

P. Zatko. Psychological security traps. In A. Oram and J. Viega, editors, *Beatiful Security*, chapter 1, pages 1–20. O'Reilly Media Inc, 2009.

K. Zelonis. Avoiding the cyber pandemic: A public health approach to preventing malware propagation, 2004.

Y. Zhu and Y. Hu. SNARE: A strong security scheme for network-attached storage. In *22nd International Symposium on Reliable Distributed Systems*, page 250, 2003.

D. A. Zilberbrand. *Efficient Hybrid Algorithms for Plan Recognition and Detection of Suspicious and Anomalous Behavior*. PhD thesis, Bar-Ilan University, Mar. 2009.

B. H. Zou, J. Zhu, and T. Hastie. *New Multi-Category Boosting Algorithms Based on Multi-Category Fisherconsistent Losses*. 2008.