NTNU

Norwegian University of
Science and Technology

Detection of intermediary hosts through TCP latency propagation

Gurvinder Singh

Master in Security and Mobile Computing

Submission date: June 2009

Supervisor: Stig Frode Mjølsnes, ITEM

Co-supervisor: Svein Willassen, Svein Willassen AS

Sasu Tarkoma, TKK, Finland

Norwegian University of Science and Technology
Department of Telematics

Problem Description

TCP deals with network latency automatically, by limiting the amount of unacked data that can be sent. As a result, a receiver will receive data in bursts, with pauses in between. The length of the bursts and the pauses depends on the network latency as well as other factors. These lengths will to a certain degree propagate through intermediary hosts. Thus, it should be possible for a receiver to decide if the other end of the TCP-connection is the original sender or a proxy, by analyzing the network latency implied by the data bursts.

The taskis to investigate if this is possible in practice, under which circumstances it is possible, and how far reaching conclusions can be made at the receiving end. Possible applications for the scheme should be outlined. Can the scheme for instance be used to fight spam, or to assist in automated tracing of computer in
trusions?

Assignment given: 19. January 2009
Supervisor: Stig Frode Mjølsnes, ITEM

Acknowledgements

I would like to express my gratitude towards those who made this thesis possible, those who supported me during the work as well as those with whom it has been a true pleasure to work.

I would like to express my sincere thanks to Prof. Stig F. Mjølsnes from Norwegian University of Science and Technology (NTNU), an admirable professor, and an expert in the field of information security for his great help. His support and encouragements together with the interesting and helpful discussions are sincerely acknowledged.

I am indebted to my supervisor Dr. Svein Y. Willassen (Wilassen AS) for guiding the work on this challenging research topic. His suggestions and valuable guidance made this study possible. Prof. Sasu Tarkoma from Helsinki University of Technology, for his support during this thesis work.

I would like to express my gratitude towards my family and Katja for their moral support. I would like to thank Erasmus-Mundus commission for the financial help during master studies. I would also like to thank Lars & Lars for our pleasant discussions during the lunch and coffee breaks at the work place.

Finally yet importantly, thanks to GOD who make all things possible, who put me in the right place at the right time and provided me the right knowledge to make this work possible.

Trondheim, 2009

Gurvinder Singh

Abstract

Today people from all lifestyles, government officials, researchers and executives use internet. The people start to depend on internet for their daily life. However, the increased dependence comes with a great risk. The popularity and potential of internet attracts users with illegal intentions as well.

The attackers generally establish a connection chain by logging in to a number of intermediary hosts before launching an attack at the victim host. These intermediary hosts are called as stepping-stones. On the victim side, it becomes hard to detect that the peer communicating with the victim is whether a real originator of the connection or it is merely acting as an intermediary host in the connection chain.

This master dissertation proposed an approach based on *Interarrival packet time* to distinguish an incoming connection from a connection coming via some intermediary hosts. The proposed approach uses information available at the receiving end and applicable to encrypted traffic too. The approach was successfully tested for SSH, Telnet, FTP, HTTP and SMTP protocols and implemented in to an intrusion detection system for corresponding protocols.

The main applications for the proposed approach are Manual intrusion detection, Tor usage detection and Spam messages detection. The approach is also applicable for the digital forensics investigations.

Keywords : Network security, Stepping stone detection, Manual intrusion detection, Tor usage detection, Spam detection and Digital forensics investigation.

Contents

List of Figures

List of Abbreviations

Telnet	Teletype network
SSH	Secure Shell
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
SMTP	Simple Mail Transfer Protocol
IPT	Interarrival Packet Time
rwnd	Receiver Window
cwnd	Congestion Window
MSS	Maximum Segment Size
RTT	Round Trip Time
RTO	Retransmission Timeout
IW	Initial Window
RW	Restart Window
TG	Traffic Generator

"Three Rules of Work:
Out of clutter find simplicity
From discord find harmony
In the middle of difficulty
lies opportunity."

Albert Einstein

1 Introduction

1.1 Motivation

Internet is one of the most significant inventions in history of the mankind. When internet was developed in 1970s, it was supposed to use only by government and scientific communities. Today it is used by people working in all sectors, from government, scientific to e-commerce sectors. It is a valuable information resource and provides wide spectrum of features to its users which allures them.

There is a trend in current society that where there is something of potential value, there is a potential of crime too. Cyber attacks on nation infrastructure, like military or commercial system, are an example of potential cyber attacks. In 2003, Slammer worm [28] infected Ohio state nuclear power plant and disabled security monitoring system. In 2007, the world's first cyber terrorism attack on Estonia [26] targeted government, credit card system and social service websites, and caused nationwide denial of service attack. In 2009, Conflicker worm [35] has infected millions of systems worldwide and created a large scale botnet to send spam. These are some of the recent examples of attacks launched with highly sophisticated technology to cause disruption on large scale.

Recently there is a rising demand for the user privacy on internet. The Onion Router (Tor) [48] is a framework of virtual tunnel which provides anonymous communication to improve privacy on internet. The Tor middle nodes (by default 3 nodes) in the network path between client and server works as proxy for client and provide anonymous communication. But due to anonymous communication, it attracts attackers who use Tor to launch attacks and hide their true identities. The study conducted by Damon et. al [38] suggested that Tor is mis-used mainly for copyright infringement, hacking attempts and web page defacement.

Another growing problem on internet is of unsolicited email message called spam. On internet approximately 90-95% [1] email messages are spam and every day 30

billion messages are sent through internet. The problem of spam is their unwanted use of network resources and their use to harm recipients by phishing scams or malware infections. Ramachandran et al. [45] studied the network level characteristics of spam messages and found out the source of most spam message is botnets. The bots in botnets work as proxy for spammers and help them to hide their true addresses.

The number of network based attacks and their scale of disruption is growing because attackers can very easily hide their true network identities and thereby reduce their chances of being captured and punished. The attackers relay their attacks through intermediary hosts/proxy system to hide their true addresses. On the receiver side, it becomes hard to detect whether peer communicating with the receiving host is a real originator of a connection or it is merely relaying the connection. In digital investigation this uncertainty causes the peer communicating with receiver to become victim of suspicion for launching the attack. If the presence of an intermediary host in a connection can be detected at the receiving end, then the mentioned attacks can be stopped at early stages by close monitoring such suspicious connections and taking actions accordingly.

The objective of this thesis work is to investigate the network latency information at receiving end and propose a method to detect the presence of intermediary host in the connection using the available information. In this thesis work, the term intermediary host and proxy system are used interchangeably.

1.2 Problem Description

Transmission Control Protocol (TCP) uses flow control and congestion control algorithms to control the amount of data transmitted to receiver. These algorithms deal with different network latency conditions automatically by limiting the amount of unacked data that can be injected in to the network. This result in arrivals of data bursts with pauses in between at the receiver end.

The lengths of these data bursts and paused delays depend on the network latency and other factors such as the application protocol (Telnet, SSH, SMTP or HTTP etc.) used in the communication. These lengths, to a certain degree, will propagate through intermediary hosts. The task of this thesis work is to detect, if possible, the peer communicating with local host is the real originator of a connection or it is merely acting as a proxy in the connection chain. Whether detection can be done by analyzing the network latency information implied by these bursts and delay in data arrival at receiving end ?

The possible applications which can be benefited from such detection should be outlined. If possible, a scheme to fight against spam or automated computer intrusion detection should be outlined.

1.3 Research Methodology

The abusive usage of intermediary hosts by attackers was addressed by Staniford-Chen and Heberlein [50]. They address the problem of abusing intermediary hosts called *stepping stone* in interactive sessions using *telnet*. After the proposal of Staniford-Chen, several approaches [21, 27, 52, 57, 58] have been proposed to detect encrypted stepping stone attacks. All these proposed approaches depend upon information available from many monitoring points deployed in the network to detect the stepping stones in a connection chain.

There are some active methods to detect the presence of intermediary hosts by making the client system to execute some code. The tool in [10] downloads a Java applet included in web page on the client machine and this applet sends network information such as local machine IP address back to the server. The tool in [5] uses the style sheet included within a web page to detect the Tor usage. Some more methods to detect the proxy usage from HTTP *GET* request header fields are listed in [11].

Most of the methodologies devised to fight against spam messages are based upon content inspection. The filtering of spam messages based on content inspection successfully reduces the amount of spam that actually reaches up to a user's account. One of the main tool, which employed this methodology is SpamAssassin [13]. In addition to the content based inspection schemes, many email message filters also perform address lookups to determine whether the sender IP address is in *Blacklist* [33] or not. But the problem with these approaches is that the spammer can change the message contents easily and bypass filtering. The study conducted in [45] suggested that mostly spam messages are sent by botnets with many short lived IP address systems which reduces the blacklists effectiveness.

The approach proposed in this thesis work monitors connection at the receiving end. The monitoring process records the *Inter Arrival Packet Time* of incoming packets from the sender and based on this information a decision is made about presence of intermediary hosts in the connection. The approach can be used to detect stepping stones in the case of telnet/ssh, to detect presence of Tor nodes in the case of HTTP or to detect the proxy used in relaying spam. Our approach does

not rely on packet contents, packet size or number of packets and thus applicable to encrypted traffic.

1.4 Scope of Research Work

This thesis work tries to detect the presence of intermediary hosts/proxy system in an incoming connection at the receiving end. The approach proposed in this thesis work makes no attempt to trace the connection source to its real origin point. The proposed approach tries to detect the use of an intermediary host, when delay between an intermediary host and the real originator is considerable. The use of application level gateways/firewalls will not cause a false alarm for this approach due to a very small delay on connection between the original source and the gateway.

TCP behaves differently under different network delay conditions. This result in the variation of data bursts and length of delay pauses at the receiving end. The length of data bursts and pauses also depends on the application protocols used for communication. This thesis work studies Telnet, SSH, FTP, HTTP and SMTP application protocols. The decision to choose these protocols is based on their widespread use. Also, these protocols mainly use a proxy either due to some restriction in an organization for remote access from internet or by attackers to hide their true identities.

1.5 Document Structure

The thesis report consists of six chapters.

- **Background and Tools Used:** This chapter specifies the essential TCP behavior for this thesis work. The available related work is outlined in this chapter. The tools used to perform the experiment and investigate the TCP behavior are also explained in the chapter.

- **The Technical Approach and System Model:** This chapter explains the evolution process of the proposed approach. This chapter describes the system model built for the proposed approach.

- **The Remote Login Protocols:** This chapter describes the behavior of the remote login protocols Telnet and SSH. The protocols are studied under different network delay conditions. The results obtained from the experiments are analyzed and discussed. The implementation of the proposed approach for these protocols is also presented there.

- **The FTP and HTTP Protocol:** This chapter explains the FTP and HTTP behavior under different network delay conditions. The results, discussion and possible applications are outlined. The implementation of the proposed approach for these protocols is also presented.
- **The SMTP Protocol:** This chapter studies the SMTP behavior under different network delay conditions. The results, discussion and possible applications are specified. The implementation of the proposed approach for SMTP protocol is also presented in the chapter.
- **Conclusion and Future Work:** This chapter summarizes the results and findings, specifies some future work and concludes this thesis work.

All the data files captured during experiments are made available online with this report. The implementation code is provided in appendix A. The scripts used to perform experiments are also made available online with this report.

"Freedom from the desire for an answer is essential to the understanding of a problem."

Jiddu Krishnamurti

2 Background and Tools Used

2.1 The Essential TCP Behavior

Transmission Control Protocol (TCP) is one of the main transport protocols used by Internet to transport the data from one system to another system. TCP operates at a *transport* layer and controls the data transmission between two end systems e.g. a web server and a web client. TCP provides a reliable and ordered delivery of a byte stream from one application running on a local system to another application running on a remote system. TCP uses a connection established between a client and a server using 3-way handshake to transfer data between them. In TCP each byte has a *sequence number* assigned to it which identify the order of data sent from each computer, so that the data can be transferred reliably and in order regardless of ant packet loss, fragmentation or disordering on the network path up to the receiver. TCP uses a *cumulative acknowledgment* scheme, in which the receiver sends an acknowledgment to a sender telling that the receiver has received all the data preceding to this acknowledgment number. The sequence number and acknowledgment number helps the TCP end systems to discard duplicate packets, to retransmit lost packets or to perform the ordering in out of order packet arrival.

To control the rate at which data will be transferred between the end systems, TCP uses a *sliding window* flow control protocol. In each TCP segment the receiver specifies the *receiver window (rwnd)*, suggesting the amount of data it can buffer for this connection. To avoid sending small size segments either due to small rwnd size or less amount of data emitted by the application e.g. telnet, TCP sender uses *Nagle Algorithm* [39]. The Nagle algorithm inhibits the sender from sending the new TCP segments if any previously transmitted data on a connection remains unacknowledged. The new TCP data can be sent immediately in two cases. Firstly, if the TCP connection is just established or it was idle for some time. Secondly, if the available window size > Maximum Segment Size (MSS) and size of data to be sent > MSS. Otherwise the transfer of new TCP segment will wait until an ACK of previously sent segments has been arrived.

The TCP sender uses congestion control algorithms to avoid congestion collapse in the network. These congestion control algorithms control the amount of outgoing data being injected into the network. The TCP sender uses *slow start* and *congestion avoidance* algorithms [18] for this purpose. These algorithms make use of two variables *congestion window (cwnd)*, the sender side limit on transmitting data, and *slow start threshold (ssthresh)*, a variable to determine whether to use slow start or congestion avoidance algorithm. After the handshake is completed, the TCP sender uses Initial Window(IW), initial value of cwnd, to determine the amount of data to be injected in to the network. From [18], IW size is defined as `2*MSS bytes`, but some non-standard experimental TCP extensions [17] defines IW as

```
If (MSS <= 1095 bytes)
   then win <= 4 * MSS;
If (1095 bytes < MSS < 2190 bytes)
   then win <= 4380;
If (2190 bytes <= MSS)
   then win <= 2 * MSS;
```

Where MSS is allowed Maximum Segment Size on the given network path. The sender sets the initial value of ssthresh equal to the size of advertised window. During the slow start, a TCP sender will increment the cwnd by MSS on arrival of each non-duplicate ACK. The slow start ends when cwnd exceeds ssthresh or when congestion is observed. During congestion avoidance, cwnd is incremented by only one MSS bytes on each RTT.

When the packet loss occurs on the connection, a TCP sender can detect it by either timeout of its retransmission timer or arrival of 3 duplicate ACKs from the receiver. In case of retransmission timer timeout (RTO), the value of cwnd is set to loss window, which is equal to 1 MSS. whereas in other case, the TCP retransmits lost segment without waiting for RTO to expire and uses fast retransmit algorithm. After retransmission, the fast recovery governs data transfer until non-duplicate ACK arrives. After this, the cwnd is set to ssthresh.

```
ssthresh = max (FlightSize/2, 2*MSS)
```

where Flightsize is amount of outstanding data in the network. Then the TCP sender will transmit data according to cwnd value by using the congestion avoidance. If TCP has not received a segment for more than one RTO then the connection is assumed to be idle and cwnd is reduced to the value of Restart Window (RW) which is $RW = min(IW, cwnd)$. On this idle connection, the data can be sent to receiver using this cwnd value as soon as it arrives from the user.

An important inference can be drawn from the above discussion regarding the TCP behavior. Initially the TCP sender has cwnd of size 2-3 MSS, the sender will send 2-3 segments to receiver and wait for ACK to arrive. The receiver will receive these segments after half of the RTT delay on a connection and upon receiving these segments receiver will send an ACK back to the sender. The sender will receive the ACK after waiting time equal to one complete RTT delay of the connection. After the ACK arrival, sender will increment the cwnd size and send more segments to the receiver. Due to the wait for ACK at the sender side, the next TCP segment at receiver end will arrive after a gap of one complete RTT. The same phenomenon will repeat itself at receiver side but with a diminishing size of time gap. The arrival of ACK at sender will result in an increment in cwnd size. Therefore, more and more segments will be sent before waiting for an ACK. This causes the new segment to arrive at the receiving end while the sender is still sending segments allowed by the cwnd size. The receiver will send an ACK for arrived segments to the sender, which may arrive at the sender while it is sending segments or after small waiting time compare to RTT.

If there is any packet loss in the network, it will cause cwnd to reduce its size. This will result in a larger waiting time gap as described above at the receiving end. This behavior will preserve itself even if the connection is a part of a connection chain established up to the receiver via some proxies in the network path. On receiver end, the packets will arrive after a gap of maximum RTT delay on any connection in the connection chain up to the receiver. If RTT of a connection between a receiver and a peer communicating with it is small compared to RTT on any connection behind the peer in connection chain, then this will be noticeable at the receiving end.

2.2 Related Work

The abusive usage of intermediary hosts by attackers is first analyzed by Staniford-Chen and Heberlein [50]. They addressed the problem of abusing intermediary hosts in the interactive sessions using *Telnet*. In the interactive sessions, the attacker launches an attack by logging into a remote system and then logging into another remote system and thus making a connection chain up to the victim system. They defined the problem of detecting stepping-stones to trace the identity of attacker using traffic contents. They proposed to use the contents thumbprints to compare two different flows around the same time on the network. The thumbprints are used to find a correlation between flows. If two flows have almost the same thumbprints then they are most likely to be a part of the same connection chain. The idea relies mainly on the traffic contents and therefore is not applicable to the encrypted traffic.

After the proposal of Staniford-Chen, several authors have proposed methods to detect encrypted stepping stone attacks. The first approach to use the timing information of a connection to trace the stepping stones was proposed by Zhang and Paxon [58]. In their approach, they calculated the correlation of different flows based on OFF periods of the each flow. A period is considered an OFF period, if there is no data traffic in the flow for more than a specific time period threshold. Their approach is based on the observation that if two flows are in the same connection chain then their OFF period coincides with each other. The scheme mainly requires that the flows are synchronized.

Yoda and Etoh [57] proposed a method to detect the stepping stone connection based on deviation of two different flows. The deviation is defined as the difference between average propagation delay and minimum propagation delay between two flows. The approach makes use of monitoring points deployed in Internet to monitor the connections and log in to records. These logged records will be used to compare with records from the compromised host. The observation behind this scheme is that the deviation of two uncorrelated flow will be large enough to differentiate it from the deviation of correlated flows in same connection chain.

Wang *et al.* [52] proposed a correlation scheme using Inter-Packet Delay (IPD) time of a connection to detect stepping stones. The scheme is based on the observation that in an interactive session, incoming and outgoing flow from a stepping stone are correlated. The correlation metric is defined over IPDs in a sliding window of packets of connections to be correlated. They suggested that the IPD of interactive sessions like Telnet, SSH or Rlogin are preserved across many hops and stepping stones. They have shown that their method works well if connections on different paths have distinctive timing characteristics.

The drawback of the above mentioned approaches is that they are vulnerable to active timing perturbation by the attacker. When timing perturbation is added to the flows, timing information used in the deviation and IPD calculation may be destroyed. The correlation between OFF periods will also be destroyed and result in the high false negative rates. Donoh et al. [27] indicated that there are theoretical limits on the ability of attackers to evade detection using timing or chaff perturbations for the sufficiently long connections. They assumed that attacker has maximum delay tolerance. They used wavelets and similar multiscale methods to separate the short-term behavior of the flows from the long-term behavior. Their method requires the intrusion connections to remain in connected state for longer periods.

Wang and Reeves [52] presented an active watermarking scheme, which is de-

signed to be robust against timing perturbations. They introduced watermark into a connection by slightly adjusting packet interarrival times of the selected packets in a flow. The watermark introduced is used to correlate the independent and identically distributed perturbation. This scheme suggested that if the timing perturbation were not large enough, the watermark information would remain along the connection chain.

The aforementioned approaches mainly use the timing related information of a connection to detect the stepping stones in the connection chain. A number of advanced approaches was also proposed to address dummy packets (called *chaff*) that can be added by an attacker, in addition to timing perturbations. These approaches provided a lower bounds on amount of chaff required to evade detection [21] and upper bounds on the number of packets required for confident detection [59], and compare tradeoffs among detection rates, false positive rate and computation cost [43]. However, all these schemes use many monitoring points deployed in the network to perform calculation of the deviation or correlation to detect the stepping stones.

The approach proposed in the present thesis work monitors incoming connection only at the *receiving end* and uses interarrival packet time to detect the presence of intermediary hosts. The intermediary host can be acting like a stepping stone in case of the interactive session or any proxy used under SMTP, HTTP or FTP session. The scheme does not try to trace the connection to its real origin point but tries to detect the presence of intermediary hosts in an incoming connection.

2.3 Experimentation Tools

2.3.1 Testbed

A testbed was needed to perform the experiments for thesis work. There are some publicly available testbeds and they are classified by [12] into four categories: cluster testbeds, overlay testbeds, federated testbeds and networking research kits. The Emulab, a component of Netbed [53], is an example of the clustered testbed. The overlay testbed examples are DARTnet (DARPA Research Testbed network) [16], the wide area component of Netbed [53], Planetlab [20] etc. The federated testbed can be created by interconnecting the several available research testbeds. The ATM switches kit distributed by Washington University [51] is an example of networking research kits. The DETER (Cyber Defense Technology Experimental Research) testbed [6] is designed to provide an experimental testbed for the cyber security research and develop mitigations mechanisms against the new threats. In the DE-

TER, a user can remotely log in and allocate the network resources, load disk images and start experimentation. The LARIAT (Lincoln Adaptable Real-time Information Assurance Testbed) testbed [49] is an extension to DARPA's intrusion detection testbed. The LARIAT supports real-time, automated and quantitative evaluations of the ID systems with script generated background traffic and real network attacks.

Considering the amount of work needed and available time for the present thesis work, a Testbed is created at Department of Telematics, NTNU, Trondheim using three computer systems. One system is used as a *Server S*, second system is used as a *Proxy P* and third system is used as a *Client C*. All the three computer systems are running `Ubuntu 8.10` operating system with `Kernel 2.6.27-11-generic`. All the systems are connected to ethernet LAN with 100Mbps capacity. The RTT delay between the server and client system under normal conditions is less than 1 millisecond(ms).

2.3.2 TG: The Traffic Generator

The TG [14] program generates and receives one way traffic streams between the traffic source and traffic sink nodes in the network. TG program is controlled by a simple language, which specifies its operating mode, protocols, addressing function and different traffic parameter. The TG program can either work as a traffic client mode in which it sends data packets or as a traffic server mode in which it works as a sink for the received traffic and sends an ACK in case of TCP traffic. TG supports TCP and UDP transport protocols with unicast and multicast (UDP only) addressing. TG can generate traffic with the several packet length and interarrival packet time distributions e.g. exponential, uniform, markov etc. An example to generate TCP traffic and send it to the TG server on host 129.241.208.198 at port 1234 is shown below.

```
on 0:5
tcp 129.241.208.198.1234
at 5 setup
at 6 arrival exponential 0.1 length exponential 576
seed 321423 time 10
```

The above configuration specifies that TG will wait for 5 seconds before initiating connection setup to host `129.241.208.198`. At 5 seconds, it will start a TCP connection setup with interarrival packet time using exponential distribution with mean of 0.1 seconds and segment length with mean of 576 bytes. The last line specifies seed value used by distribution to generate a random value and time 10 specified that the communication will last for 10 seconds after the setup.

2.3.3 Netcat: The GNU Netcat

The Netcat [7] is a simple UNIX utility, which can read and write data across the network connections using TCP and UDP transport protocols. It is designed to work as a back-end tool that can be used directly or invoked by other programs and scripts. Some of its main features are as:

- Outbound and inbound TCP or UDP connections on any ports.
- Act as a general TCP proxy without considering application.
- Support for buffered send-mode (one line after every N seconds), hex dump of transmitted or received data.

In the present thesis work, netcat is used to work mainly either listening at a specific port for the incoming connection or working as a proxy for a TCP connection. Below an example shows the netcat working as a proxy

```
$ mknod backpipe p
$ nc -l -v -p 1234 0<backpipe | nc 129.241.209.234 1234 1>backpipe
```

The first command creates a pipe on local file system to carry the data in both directions. The second command makes the local host to listen for a connection at port 1234. The received data on this port is forwarded to the host `129.241.208.198` at port 1234 and response is sent back to the client with help of a newly created pipe.

2.3.4 Netem: The Network Emulator

To create the controlled Internet behavior in the laboratory, *Netem* [31] coupled with *Traffic Control (tc)* tool is used. The both Netem and traffic control tool are a part of *iproute*2 package of Linux kernel. The traffic control tool uses Netem to emulate Internet behavior. Three main network emulators are available Netem, DummyNet [47] and NIST Net [22]. DummyNet is an integrated part of FreeBSD and is implemented as a part of packet filtering mechanism. DummyNet operates on outgoing packets, which is similar to the Netem. The NIST net is a Linux kernel extension and offers complex delay, loss and other emulation scenarios. The NIST Net operates on incoming packets before they reach at protocol stack and uses a high resolution timing source to perform its task. The NIST net is complex to use and configure. The Netem uses most of NIST Net functionality and it is a part of Linux kernel from version 2.6 onwards.

The Netem uses existing Quality of Service (QoS) and Differentiated Service (DiffServ) facilities in Linux kernel. Netem uses command line tool *tc* for the configuration and a small kernel module for queuing discipline. The basic queuing architecture of Linux kernel is shown in Figure 2.1. The queuing discipline sits between the protocol output and the network device. The queuing discipline is an object with two key interfaces. One interface receives packet from IP protocol and another interface forwards these packets to the network device. The queuing discipline makes the decision of forwarding packet based upon the defined policies. The Linux kernel provides a rich set of disciplines for queuing, prioritization and rate control policies. The default queuing discipline is FIFO queue but it can be replaced with Priority queue, Token Buffer Filter (TBF) or Random Exponential Drop (RED). The Priority queue is used in experiments performed during this thesis work. The Priority queue is a classful queuing discipline and provides functionality to filter packet based on flow id, destination address or other packet fields.

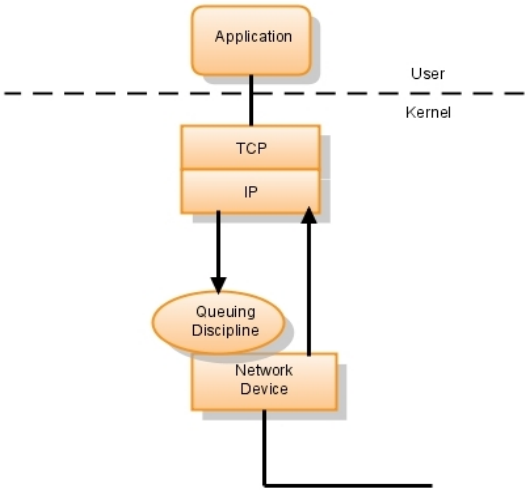

Figure 2.1: Linux Queuing Architecture

The Netem can add delay in network using various statistical distributions, can cause a packet loss using statistical distributions or can do packet duplication. It can perform packet re-ordering or corruption of every *nth* packet with given probability percentage. An example is given below, which change queuing discipline for device *eth0* to priority queue and then attaches Netem to priority 3 queue with 20ms delay using normal distribution. This rule is applied to packets, which are destined to IP address `129.241.209.123`.

```
# tc qdisc add dev eth0 root handle 1: prio
# tc qdisc add dev eth0 parent 1:3 handle 30: netem \
  delay 20ms 2ms distribution normal
# tc filter add dev eth0 parent 1:0 protocol ip prio 3 \
  u32 match ip dst 129.241.209.123 flowid 10:3
```

2.3.5 Daemonlogger: The Packet Sniffer

The *Daemonlogger* [4] is a packet sniffer tool to sniff live traffic from a given interface. The daemonlogger captures the incoming packets based on the given filter applied during startup and store the captured packets in *libpcap* [9] file. The Linux kernel uses different method to handle packet capturing stack. It uses queue instead of buffer to store the capturing stack with pointers behind the applied filter. The Linux kernel can cause a packet drop during capturing traffic at a high speed. To increase the kernel capabilities and allow it to capture packet at high speed without causing any packet drop, there is a need to increase the amount of memory space allocated to kernel capturing engine. This allows kernel to handle incoming packets on high speed connection without causing any degradation in capturing performance. This is done by changing the Linux kernel parameter in */proc* file system [8].

```
echo 33554432 > /proc/sys/net/core/rmem_default
echo 33554432 > /proc/sys/net/core/rmem_max
echo 10000 > /proc/sys/net/core/netdev_max_backlog
```

From [29], allocating 32MB size of memory to Linux kernel capturing engine is enough to capture packet up to 650 MBit/sec speed without causing any performance degradation. The packets are captured from the monitoring system in the present thesis work. The monitoring system can be either the *server S* or the *client C* depending upon nature of the experiment.

2.3.6 Wireshark: The Packet Analyzer

The *Wireshark* [15] is a packet analyzer application, which understands the structure of different networking protocols. It is able to dissect the captured packets in to corresponding network protocol fields and provide their meaning along with detailed information. The Wireshark provides various options to analyze captured packet more efficiently and can capture live packets. It supports various capture and display filters to show the required information in a user friendly way. An example is shown below to see the interarrival packet time of HTTP *GET* request from the host `129.241.209.234`.

```
filter: ip.src==129.241.209.234 and http.request
display filter: frame.time_delta
```

The Wireshark has a capability to show the protocol specific statistics information of the captured network session and draw corresponding graphs. It provides facility to copy the required information in CSV format and enable user to perform complex tasks on it.

2.3.7 Expect: The Automating Tool

The Expect [36] is a tool which automates the interactive applications such as SSH, Telnet, FTP etc. It reads a script, which contains commands similar to dialogue with a user and includes multiple paths depending upon the application response. The Expect is used to produce reproducibility in typing behavior. It is not possible for a human user to produce the exactly same typing behavior during more than one test. Therefore, Expect provide consistency in typing behavior and delay under all tests. The script contains:

- **Send** - This directs Expect to send the command to application as it is typed by user himself.
- **Expect** - After sending command to application, Expect waits for response matching a given expression with *Expect* directive and take action accordingly.
- **Control Sequence** (if/then/else/while etc.) - Expect supports high level control flow and allows to take different action depends upon expression evaluation.
- **Job Control** - Expects enables control of more than one application simultaneously and allows execution of UNIX programs.
- **User Interaction** - Expects enables user to take control back during script execution and returns control back to the Expect.

The Expect scripts are written in high level procedural language with elements taken from C, LISP and Tcl. An example below shows how to start SSH session to given username, password and IP address.

```
# set the command line parameters to variables
  set user [lrange $argv 0 0]
  set password [lrange $argv 1 1]
  set ipaddr [lrange $argv 2 2]

# spawn the shell with ssh
  spawn ssh $user@$ipaddr
```

```
# check if host is communicating with target host for first
# time then send yes and store target host's public key
# "*Connection refused*" {puts "Connection refused by
# Remote Host::$ipaddr"; exit}
  while {1} {
        expect {
               "*yes/no*" {send -- "yes\r"}
               "*$user@$ipaddr's password: "
                      {break;}
               }
       }
# Below all the sleep statements are to simulate
# the user typing behavior
  send -- "$password\r"
  expect  "*$user@*"
```

The script spawns a shell with ssh to a given host with specified username and password. It checks if the connection attempt is for the first time and then it sends yes to store peer's public key. When connection prompts for password, loop breaks itself and password is sent. After sending password, the command prompt is expected to perform further tasks.

2.3.8 BRO: The Intrusion Detection System

The network intrusion detection systems are used to detect the unauthorized access of network resources by the attackers. These systems can be divided into two categories. The first type of systems are those that rely on audit information gathered by the hosts in a network, which they trying to protect. The second type of systems are those that operate "stand alone" by monitoring the network traffic directly using a packet filter. These days hybrid systems are also getting popularity that combine the two approaches. The *Bro* [41] is a standalone system for detecting the network intruders in a real time by monitoring the network traffic.

Bro is divided in to an "event engine" which converts stream of filtered packets in to a stream of high level network events and an "interpreter" which interpreters a specialized language used to express the site's security policy. The structure of Bro system is shown in Figure 2.2. Bro uses libpcap [9] library to capture the packets and isolates itself from details of the underlying network link technology (FDDI, Ethernet, SLIP etc.). The libpcap uses available kernel packet filter BPF [37] to reduce the traffic amount entering in to the system.

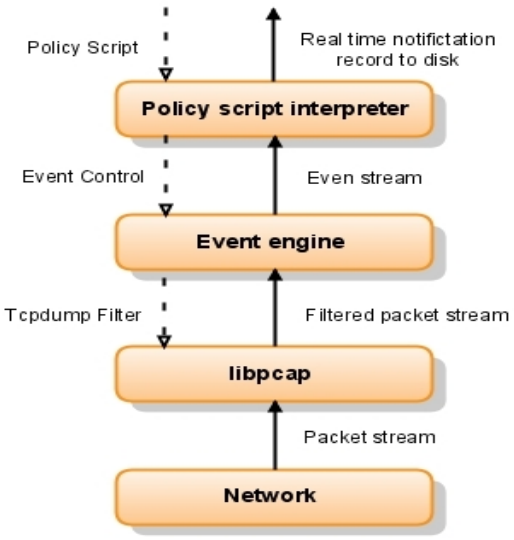

Figure 2.2: Structure of Bro IDS system

The resulting stream of filtered packets is sent to the event engine, which first perform integrity checks on the packet header and structure. If check fails, Bro generates an event indicating the cause and discards the packet. If the check succeeds then the event engine looks up the connection state corresponding to the communicating peers and TCP or UDP ports, and creates a new state if none exists. It then dispatches the packet to a handler specified by the site's security policy. The Policy script interpreter checks whether any events are generated by event engine or not. If so, it processes events in FIFO fashion until the queue becomes empty. The present thesis work uses Bro IDS to implement the proposed approach for different application protocols. Bro provides support for implementing new algorithms to detect the network intrusion.

"Everything should be made as simple as possible but not simpler."
Albert Einstein

3 The Technical Approach and System Model

3.1 The Evolution of Approach

In the TCP protocol, the amount of data transmission is limited by allowed window size (min(cwnd,rwnd)). The Nagle algorithm inhibits the sender to transmit small size data packets in order to use the network resources more effectively. These two principles make TCP to behave distinguishably in a connection with the different Round Trip Time (RTT) delay. To study the variation of TCP behavior under different network delay conditions, some tests are performed on the Testbed. The intuition behind the performed tests is that in a connection with larger RTT delay, the Nagle algorithm makes the sender wait for ACK to arrive and as a result, the sender will send large size segments to the receiver. Therefore, to transfer the same amount of data, the number of packets will decrease, as RTT delay in the communication path will increase.

3.1.1 The Packet Size and the Packet Number Approach

To study the TCP behavior, two types of experiments are performed in the laboratory. In the first experiment, TCP is used to transfer 1 MBytes of data between the client (sender) and the server (receiver) with/without presence of proxy in the network under different delay conditions. In the second experiment, the effect of different delay conditions is studied on the traffic with small size application data.

A traffic generator tool *Traffic Generator (TG)* [14] is used to generate the TCP traffic using an exponential distribution for the interarrival packet time and the packet length. In these experiments, Netcat [7] is used as a proxy between the client and the server. The delay in the network path is induced with the Netem [31] using a normal distribution. The Daemonlogger [4] is used to capture packets at the server side.

TCP Data Transfer Mode Experiment

In the first experiment, the TG program is generating data using a exponential distribution with the mean interarrival time of 1ms and the mean packet length of 1500 bytes. In the first phase, the tests are performed without any proxy on the network path under the following delay conditions.

$$S \xleftrightarrow{10ms} C \tag{3.1}$$

In the second test, the delay is increased up to 100ms in order to study a change in the TCP behavior.

$$S \xleftrightarrow{100ms} C \tag{3.2}$$

In the third test, the delay is increased up to 500ms.

$$S \xleftrightarrow{500ms} C \tag{3.3}$$

In second phase of the first experiment, a proxy system is included in the network path between the server and the client. The delay between the server and the proxy system is set to 10ms and delay between the proxy and the client system varies as.

$$S \xleftrightarrow{10ms} P \xleftrightarrow{10ms} C \tag{3.4}$$

In the second test, the delay between the proxy and the client system is increased up to 100ms, in order to study the change in TCP behavior at the server end.

$$S \xleftrightarrow{10ms} P \xleftrightarrow{100ms} C \tag{3.5}$$

In the third test, the delay is increased up to 500ms.

$$S \xleftrightarrow{10ms} P \xleftrightarrow{500ms} C \tag{3.6}$$

In third phase of the first experiment, the delay between the server and the proxy system is increased up to 100ms. The delay between the proxy and the client system varies in a similar pattern as in the second phase.

$$S \xleftrightarrow{100ms} P \xleftrightarrow{10ms} C \tag{3.7}$$

In the second test, the delay between proxy and client system is increased up to 100ms, in order to study the change in TCP behavior at the server end.

$$S \xleftrightarrow{100ms} P \xleftrightarrow{100ms} C \tag{3.8}$$

In the third test, the delay is increased up to 500ms.

$$S \xleftrightarrow{10oms} P \xleftrightarrow{500ms} C \tag{3.9}$$

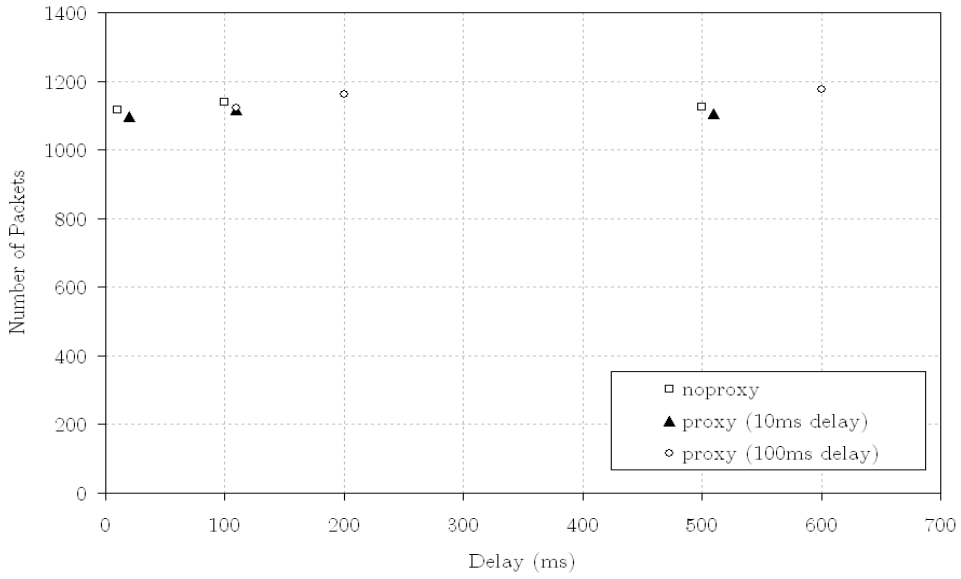

Figure 3.1: Experiment for the TCP data transfer mode under different delay conditions. The X-axis shows total RTT delay on the connection from the server to the client system and the Y-axis shows number of packets.

The number of packets is counted for with/without proxy cases in all three test phases. The results of the first experiment are shown in Figure 3.1. In Figure 3.1, the symbols marked as "noproxy" case shows the result from the first phase of experiment. The symbols marked as "proxy (10ms delay)" shows the result from the second phase and the symbols marked as "proxy (100ms delay)" shows the result from third phase of the experiment. This terminology remains same in all tests performed in this chapter.

The results in Figure 3.1 show that the number of packets is either remained almost the same or slightly increased, in spite the expectation of their decrease. The reason for this behavior is almost all segments are filled completely with data of size equal to MSS of the network path. The slight increase of the number of packets is due to increase of the number of ACK packets sent by the server upon receiving data.

The similar behavior is noted when the proxy system is present in the network path. The number of packets either remained almost the same or slightly increased with an increase in the delay. The main point to note in this experiment is when the delay between the server and the proxy system is kept same and delay between the proxy system and the client is increased, the number of packets start to increase.

However, the increase in the number of packets is not much to make a basis for the detection of a proxy system in the incoming TCP connection.

TCP Interactive Traffic Type Experiment

The second experiment is conducted to study the TCP behavior with an exponentially distributed size of data packets with mean of 10 bytes. The intuition behind this experiment is that in the first experiment, almost all segments are of MSS size and the TCP sender is transmitting data as soon as it received the data from the user. The Nagle algorithm does not inhibit a sender from sending data if the data to be sent is of MSS size and the window size is greater than MSS. However, the Nagle algorithm inhibits a sender from sending small size data packets and makes the sender to wait for ACK to arrive from the receiver. With an increase in the delay on a connection, the data will be buffered for longer time and result in arrival of packets with a larger size at the receiving end.

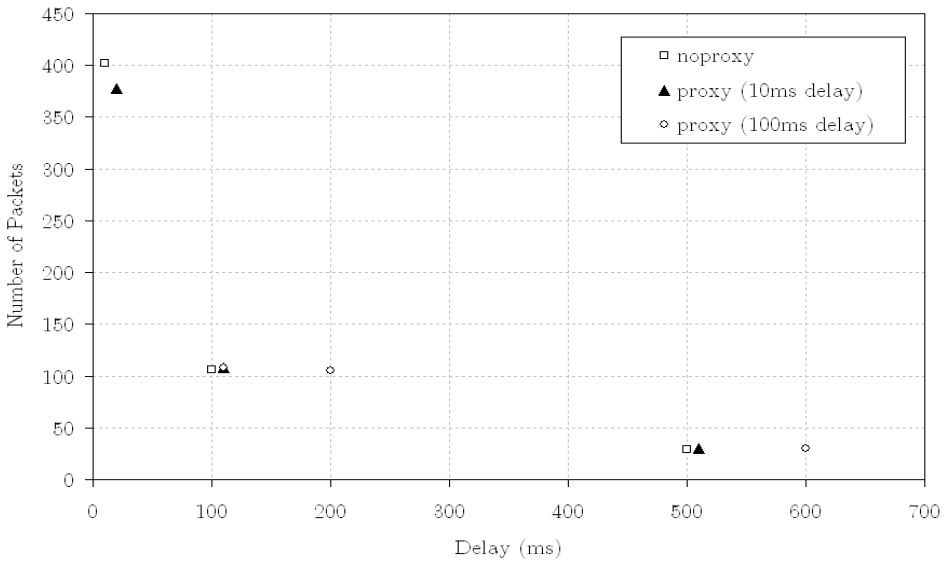

Figure 3.2: Experiment for the TCP traffic in small packet size mode under different RTT delay conditions. The X-axis shows total RTT delay on connection from the server to the client system and the Y-axis shows the number of packets.

In the second experiment, the TG program is generating data using an exponentially distributed packet size with the mean of 10 bytes. The experiment is

performed to transfer 10 Kbytes of data.

The second experiment is conducted under the similar delay and proxy conditions as in the first experiment with an exception in the data generation process. The results from the second experiment are shown in Figure 3.2. From the results, it is clearly seen that the number of packets are decreased as the delay on a connection is increased. The reason for such behavior is that the Nagle algorithm inhibits the client system to send small size data and makes it to wait for ACK to arrive. This causes data to buffer at the client side and upon ACK arrival, the client sends a larger size segment. Hence, to transmit the same amount of data the number of packets is decreased.

In the first phase of this experiment when no proxy is present in the network path, the number of packets are decreased from 402 to 29 as the delay is increased from 10ms to 500ms. Similar behavior is seen in the other two phases also, when the proxy is present in the network path.

In the "proxy (100ms delay)" case, there is no significant decrease in the number of packets when the delay in the connection between the proxy and the client system is increased from 10ms to 100ms. The reason behind this behavior is, when delay between the server and the proxy system is equal or higher than delay between the proxy and the client system, data at the proxy system will arrive within delay of the connection between the server and the proxy system. Therefore, the proxy system will forward data to the server without any further delay. Even if the delay between the proxy and the client system is increased from 10ms to 100ms, the number of packets remains the same.

The main result from the above experiment is that when the delay between the server and the proxy system is kept same, and the delay between the proxy and the client system is increased, then number of packets start to decrease as the delay increases. Therefore using the proposed approach it is possible to detect the presence of a proxy if the rate of data generation at the sender side is known. However, this is not possible in the real world. The application at the sender side can generate data at a random rate and can randomly generate more data, which will result in a larger packet size. The aforementioned approach will result in a false positive for this case. Besides, growing interest in the encryption usage for communication will result in a problem for this approach. Different encryption algorithms and keys used for the encryption will result in arrival of packets with the different length at the receiving end, and will result in more false positives. Therefore an another approach is needed which will be independent of packet contents, the packet size and packet numbers. The new approach should also be applicable to the encrypted traffic.

3.1.2 The Interarrival Packet Time Approach

The quest of finding a new approach directs all attention to the traffic parameters, which are independent of encryption. The required parameter turned out to be *Interarrival Packet Time (IPT)*. The IPT is a difference between the current arrival time of incoming packet from the last packet arrival time. The data from previous experiments is analyzed with different perspective to study IPT values and its behavior.

TCP Data Mode Experiment

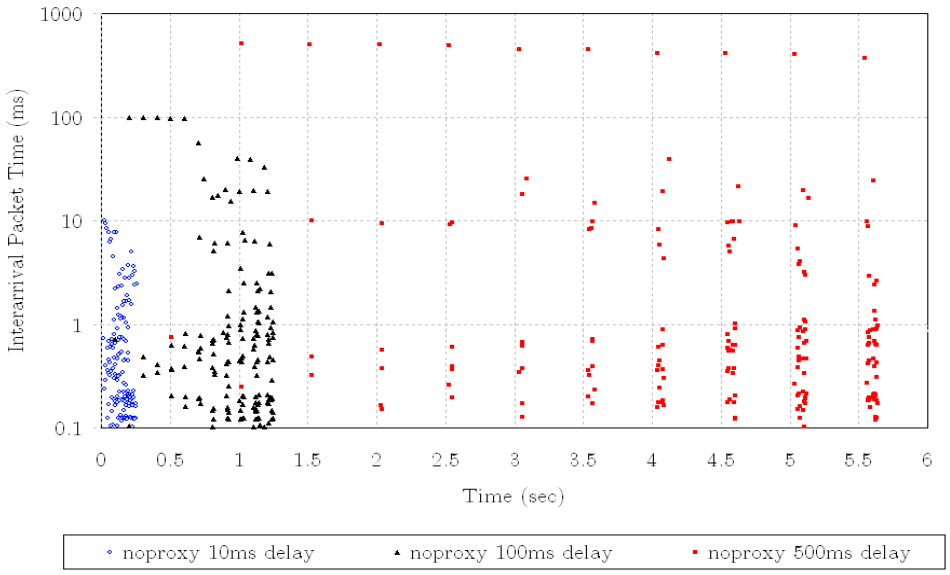

Figure 3.3: Experiment for IPT in TCP data transfer mode for the "no proxy" case under different RTT delay conditions. The X-axis shows the time (seconds) and the Y-axis shows the interarrival packet time under different delay conditions on the logarithmic scale.

Figure 3.3 shows the IPT values from the first phase of the experiment 3.1.1. There is no proxy present in the network path in this phase of the experiment. The case with "noproxy 10ms delay" shows the test without the proxy in the network path and the delay of 10ms between the server and the client system. The remaining tests use the same terminology in this chapter.

From the figure, we can see that when delay between the client and the server is 10ms, almost all packets are arrived within 10ms. In the initial phase of the data

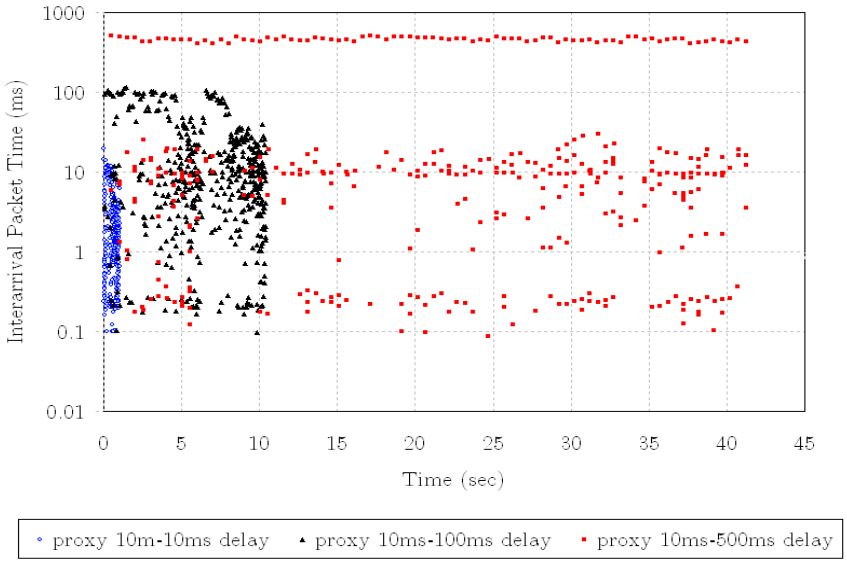

Figure 3.4: Experiment for IPT in TCP data transfer mode for the "proxy with 10ms delay" case under different RTT delay conditions. The X-axis shows the time (seconds) and the Y-axis shows the inter-arrival packet time under different delay conditions on the logarithmic scale.

transfer, TCP at the client system is in the slow start phase. After sending 2-3 packets, it waits for ACK to arrive before sending more data to the server and this results in a delay at the server end. As the data transfer progresses, the sender increases its window size and sends more data to the receiver without waiting for ACK. Therefore, we can see (Figure 3.3) the IPT value of packets starts to decrease. The other two test cases (when the delay in the network path is 100ms and 500ms) show the similar behavior.

Figure 3.4 shows IPT values from the second phase of the first experiment 3.1.1. In this phase, a proxy is present in the network path. In the second phase, the RTT delay between the server and the proxy system is 10ms and the delay between the proxy and the client system is varying from 10ms to 500ms. The case in Figure 3.4 with "proxy 10ms-10ms delay" shows the test when a proxy is present in the network path and the delay is 10ms between the server and the proxy system, and the delay in the connection between the proxy and the client system is 10ms. The same terminology is used in all the remaining tests performed with a proxy.
Consider the case when delay conditions are as

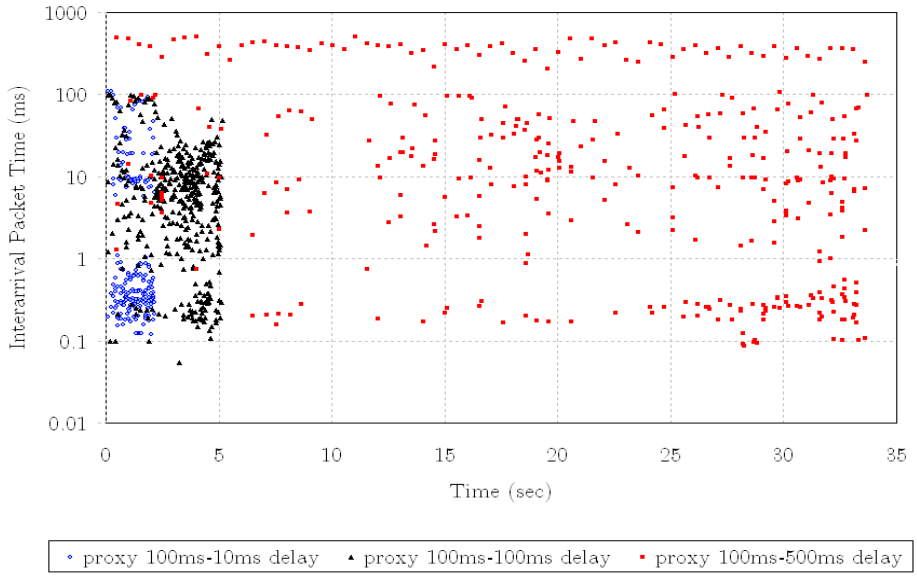

Figure 3.5: Experiment for IPT in TCP data transfer mode for the "proxy with 100ms delay" case under different RTT delay conditions. The X-axis shows the time (seconds) and the Y-axis shows the inter-arrival packet time under different delay conditions on the logarithmic scale.

$$S \overset{10ms}{\longleftrightarrow} P \overset{100ms}{\longleftrightarrow} C \tag{3.10}$$

The result shows that the packets start to arrive after 100ms delay, even though delay between the server and the proxy system is 10ms. The reason for this behavior is the small initial window size at the client system. After sending 2-3 data packets to the proxy system, the client waits for ACK to arrive before sending more data. The proxy system sends ACK back to the client and forwards received data to the server. After receiving ACK from the proxy system, the client increases its window size and sends more data to the proxy system. The new data segments arrive at the proxy system after one complete RTT delay in the connection between the proxy and the client system. The proxy system then forwards received data to the server. At the server side, this delay results in a delay equal to delay on the connection between the proxy and the client. As the data transfer progress, window size at the client side increases and allows the client to send more data without any further wait for ACK. This causes the IPT values to decrease as seen in Figure 3.4. The same behavior is noted when the delay between the proxy and the client system is 10ms and 500ms.

Figure 3.5 illustrates the result from the third phase of the experiment. In the

third phase, the RTT delay between the server and the proxy system is 100ms and the delay between the proxy and the client system is varying from 10ms to 500ms. The figure illustrates that the packets start to arrive after the 100ms delay, which is the delay between the server and the proxy system. When the delay between the proxy and the client system increases up to 500ms, the packets start to arrive after 500ms. As the window size starts to increase, IPT at the server side starts to decrease. When the delay between the proxy and the client system increases from 10ms to 100ms, the packets still arrive after 100ms. Because data arrived at the proxy system within 100ms delay which is equal to the delay between the server and the proxy system.

Therefore if the delay in a connection between the server and the proxy system is equal or higher than the delay in a connection between the proxy and the client system, then it is not possible to detect the presence of a proxy system in the network path.

TCP Interactive Traffic Type Experiment

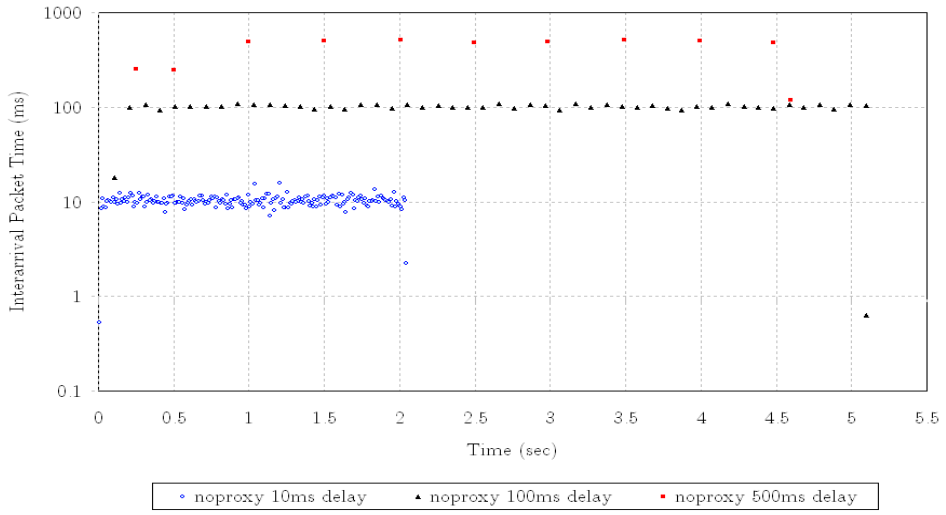

Figure 3.6: Experiment for TCP small packet size traffic with the "no proxy" case under different RTT delay conditions.The X-axis shows the time (seconds) and the Y-axis shows the interarrival packet time under different delay conditions on the logarithmic scale.

The similar analysis of IPT values is performed on the data from the second experiment. The result in Figure 3.6 shows the IPT values from the first phase of

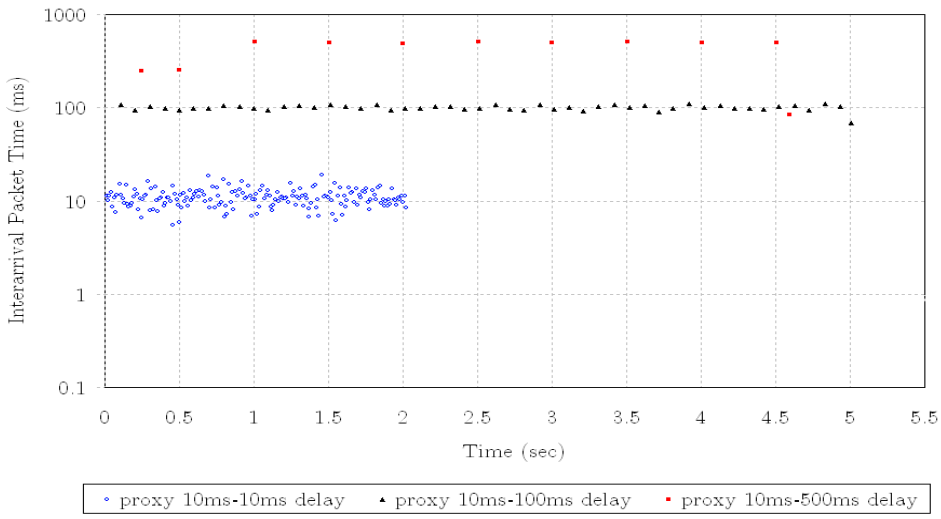

Figure 3.7: Experiment for TCP small packet size traffic for the "proxy with 10ms delay" case under different RTT delay conditions.The X-axis shows the time (seconds) and the Y-axis shows the interarrival packet time under different delay conditions on the logarithmic scale.

the second experiment. From result, we have seen that the packets arrived at the server side with in corresponding delay on the connection between the server and the proxy system. The main difference between the results obtained in first phase of the second experiment from the first experiment is the almost all the packets arrive after 10ms second delay. The IPT values do not decrease with the increase in the window size. In the second experiment, the application is generating small size data and due to small size packets, the Nagle algorithm inhibits the client system to wait for ACK to arrive before sending more data. The Nagle algorithm inhibits a sender from sending the data if the data to be sent are less than MSS in the network path. This causes a delay in the data arrival at the server end and causes average packet length to increase from 10 bytes to 48 bytes in the 10ms delay case. The similar behavior is seen for 100ms and 500ms cases.

Figure 3.7 illustrates the result from second phase of the second experiment. The delay between the server and the proxy system is 10ms and delay between the proxy and the client system varies from 10ms to 500ms. From the result, it is observed that when delay between the proxy and the client system is 100ms, the packets arrived after 100ms delay at the server end even though the delay between the server and the proxy system is 10ms. The same behavior is seen for the 500ms case and

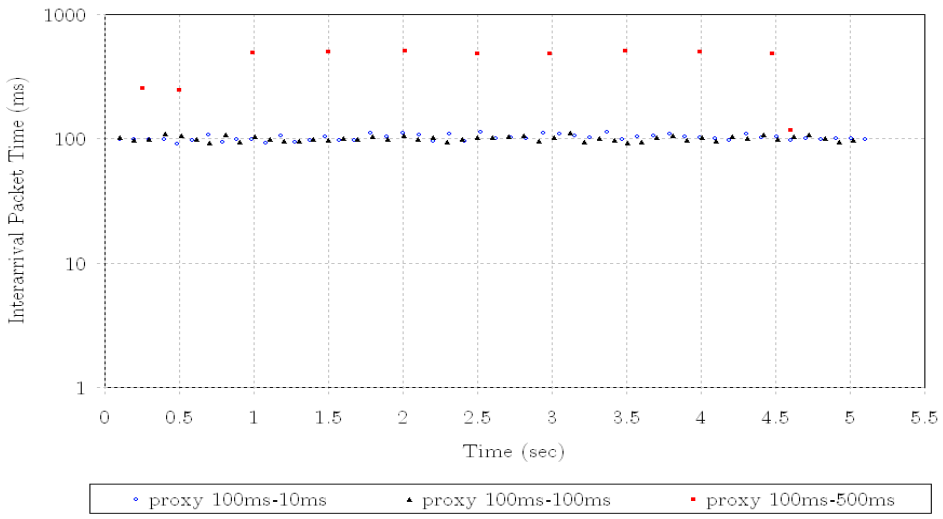

Figure 3.8: Experiment for TCP small packet size traffic for the "proxy with 10ms delay" case under different RTT delay conditions. The X-axis shows the time (seconds) and the Y-axis shows the interarrival packet time under different delay conditions on the logarithmic scale.

the value of IPT remains the same during the complete session compared to the first experiment where IPT starts to decrease due to increase in the window size. This is because the available data size in not equal to MSS size on the network path.

Figure 3.8 illustrates the result from third phase of the second experiment. The delay between the server and the proxy system is 100ms and delay between the proxy and the client system varies from 10ms to 500ms. From the result it can be seen that the packets start to arrive after 100ms at the server end, when delay between the proxy and the client system is 10ms or 100ms. This is because the data at the proxy system arrived within 100ms delay and the proxy forwards data without any further delay. However, as delay between the proxy and the client system is increased up to 500ms, the packets start to arrive after 500ms delay at the server side.

The results of IPT values obtained from the two experiments show that by monitoring the IPT values at the receiver end, it is possible to detect whether the peer communicating is a real originator of the connection or acting as a proxy in the connection. The detection is possible when delay between the receiver system and the peer communicating with it is less compared to a delay between the peer and the sender system.

3.2 The System Model

3.2.1 System Description

When a user connects from a local host H_0 to a remote host H_1 using TCP as a transport protocol, it establishes a TCP connection C_1 between the hosts H_0 and H_1. If the user connects to an another remote host using the host H_1 to a remote host H_2, it results in the establishment of another TCP connection C_2. Thus following the same procedure he successively connects to H_3, H_4,....., H_n by establishing the connections C_3, C_4,....., C_n on each link between the hosts. This sequence of connections $C = \langle C_1, C_2, ..., C_n \rangle$ is called a connection chain. The TCP connection C_i is a connection between the host H_{i-1} and the host H_i. Each connection C_i has its own RTT delay t_i between host H_{i-1} and H_i.

Problem Statement: *Given a TCP connection C_i between a host H_{i-1} and a host H_i with RTT delay t_i, find out at the receiving host H_i whether the host H_{i-1} is a real originator of the connection or it is merely acting as an intermediary host in the connection chain.*

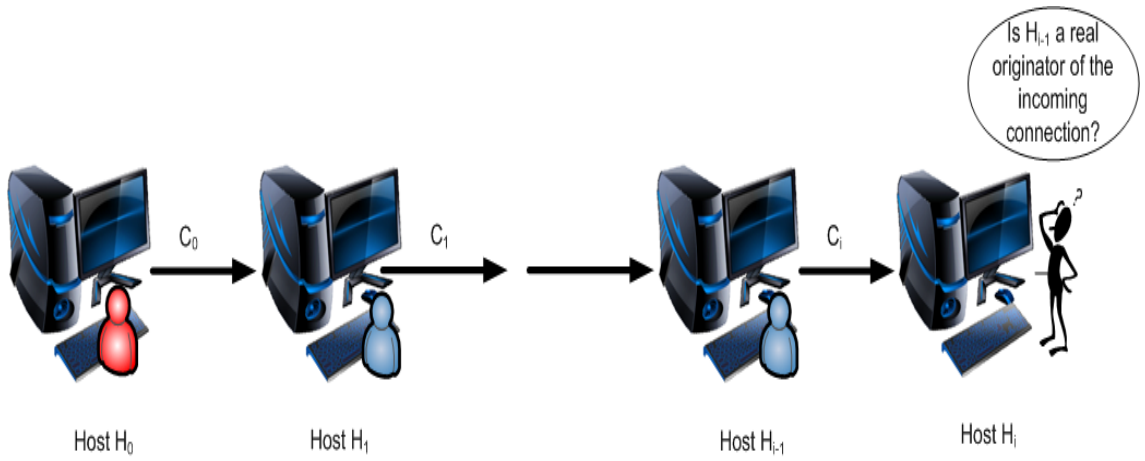

Figure 3.9: Problem Statement

3.2.2 Conditions

To solve the given problem 3.9, the proposed approach takes in account of *Interarrival packet time* of the incoming packets from the sender at the receiving end. The detection approach considers incoming packets with the packet size greater than zero. In order to apply the given approach to the encrypted communication and the

concerns about privacy issues, the approach does not take into account the packet contents and the packet size. The packet size can be changed to bypass the detection procedure and in the encrypted communication, the packet size differs due to different key sizes used for the encryption.

3.2.3 The Given Approach

The packets are coming at the receiving end with independently distributed interarrival packet time from a source. Consider a count process $Z = \{Z_i\}_{i=0,1,2,...}$ that monitors the interarrival packet time during the *ith* time interval at the receiving end. Each time interval is a bin of width b, where the value of b depends upon type of the connection being monitored. For an interactive connection type:

$$b = Avg.\ typing\ speed(TS) + RTT + Host\ load\ factor(H) \tag{3.11}$$

whereas in a data bursts connection type:

$$b = RTT + Host\ load\ factor(H) \tag{3.12}$$

The RTT is the Round Trip Time delay of the connection between the hosts and Host load factor (H) is a parameter that correspond to a delay caused by the system load to respond a request. The incoming packet size should be greater than zero to be counted in the counting process Z. For a given *ith* interval $Z_i^{(b)}$ can be, $Z_i^{(b)} = 0$ or $Z_i^{(b)} > 0$. The former case is known as a *empty* bin and later is known as a *occupied* bin. For $j \geq i$, the $Z_{i,..,j}^{(b)}$ is called as a *burst* if for all k, $i \leq k \leq j$, the bins are occupied and similarly $Z_{i,..,j}^{(b)}$ is called as a*lull* if the correspondingly bins are empty.

We are interested in the size of lulls. The size of a lull specifies the amount of period in which no packets have arrived from the sender. If the size of lull is greater than a specified *threshold* value, it means a break in typing the keystrokes by the user during remote login session. Whereas in data bursts case this is not possible under the normal network conditions and should not happen. If the size of a lull is less than a specified *threshold* value then the occurrence of such a lull during the connection is a sign of suspicion. In order to have high confidence and a low false positive rate, the intrusion detection system should wait for more occurrences of such lulls and then raise a warning, and mark the connection as a possible connection using the intermediary hosts.

There is a probability that lulls may occur due to congestion on the network path. To avoid this and decrease the probability of false positives, another count

process $Y = \{Y_i\}_{i=0,1,2,...}$ can be considered which counts the number of ACK packets arriving from the sender with the TCP segment size equal to zero. If the number of events in the process Y has been increased since the occurrence of recent lulls in the process Z, then the occurrence of lulls is unlikely to occur due to congestion. Because arrival of ACK packets from the sender signify the lack of congestion in the network path.

The RTT delay of the connection can be measured by different methods. In [19], authors suggested different methods to calculate the RTT delay of a connection with their benefits and shortcomings. In the present thesis work, the RTT delay is calculated during initial handshake between the sender and the receiver. As this is calculated immediately after the client made a request and the detection process is started in the early stage. The value of *threshold* for the lulls length can be tuned by the administrator according to his/her requirements. From the experiments performed in this thesis work, the value of *threshold* equal to 1.5 secs is found to be adequate.

The application protocols have their own traffic characteristics. In [40], the author has described empirically driven analytical models of a TCP connection under the different application protocols. In [40], the author has studied mainly the amount of bytes transfered between the server and the client and their resemblance with the various distributions. However, the author has not studied the interarrival packet time characteristics for the different application protocols. In [42], authors have studied the interarrival packet time of Telnet protocol using the Pareto distribution. But except the [42], no other literature has been found which discuss the interarrival packet time distribution for the different application protocols.

The problem in using the Pareto distribution for other application protocols is the lack of its parameters values. The value of shape parameter (β) for Telnet protocol is given in [42], but for the other application protocols no such values are provided. Therefore, due to lack of the available literature and limited time, making a complete system model is not possible in the present thesis work.

"Accept what cannot be changed, change what cannot be accepted."
Harbhajan Singh

4 The Remote Login Protocols

4.1 SSH Protocol

The Secure Shell (SSH) protocol allows exchange of data in a secure way over an insecure network. The SSH protocol was designed for a replacement of insecure remote login protocols such as telnet and rlogin. The SSH protocol consists of three main components [55]: Transport Layer Protocol, User Authentication Protocol and Connection Protocol. The transport layer protocol provides server authentication, confidentiality and integrity. The SSH transport layer protocol typically runs over TCP/IP. The user authentication protocol authenticates the client-side user to the server and runs over SSH transport layer protocol. The connection protocol multiplexes the encrypted tunnel into several logical channels and provides a wide range of services on a single tunnel.

4.1.1 SSH Experiment using Typing

The motivation for the experiment is to detect the use of intermediary hosts in an incoming connection request at the receiver end. The attacker usually logs in to a number of intermediary hosts called *stepping stones* before launching an attack at the victim system. The chain of stepping-stones helps him to hide his real IP address and remains anonymous on the network. The intuition for this experiment is that an attacker types command characters from his local system and this command characters travel from attacker system to the victim system via a chain of intermediary hosts. The journey of characters from attacker system to the victim system will result in a higher delay compared to the arrival of characters from last intermediary host in the chain.

The SSH experiment is conducted with three computer systems. The system S is acting as a SSH server, the system P is acting as a proxy system and the system C is acting as a SSH client system. The attacker is located at the client C. The proxy system P can act either as a SSH server for the client C and a SSH client for

the server S or as a generic TCP proxy using netcat. To imitate the user typing *expect* [36] scripts to provide reproducibility in all the performed experiments. The delay in typing each command character has been similar to an advance computer user, such as a hacker. The experiment uses the following command set.

- pwd
- ls -l
- who
- cd Public
- ls -l
- more g
- exit

The experiment is conducted in two phases. In first phase of the experiment, proxy system P is working as a SSH server for the client C and a SSH client for the server S. The proxy system P is not present during first test and the network conditions are as

$$S \xleftrightarrow{10ms} C \tag{4.1}$$

Now a proxy system is included in the network path between server S and client C. The network conditions on the path are as

$$S \xleftrightarrow{10ms} P \xleftrightarrow{250ms} C \tag{4.2}$$

In third test, the connection has delay up to 500ms on the connection between proxy P and client C is increased.

$$S \xleftrightarrow{10ms} P \xleftrightarrow{500ms} C \tag{4.3}$$

The Figure 4.1 shows result from this phase of the experiment. The Figure illustrates the IPT of incoming request packets from the sender. The result shows the IPT values only for packets encrypted with the current session key. These encrypted packets include username, password and the command characters.

In second phase of the experiment, the proxy system P is acting as a TCP generic proxy. The network condition and command script used in this phase is similar to the first phase of the experiment. The Figure 4.2 illustrates result from this phase. In this phase, the result shows IPT values for key exchange packets and encrypted packets coming from the sender.

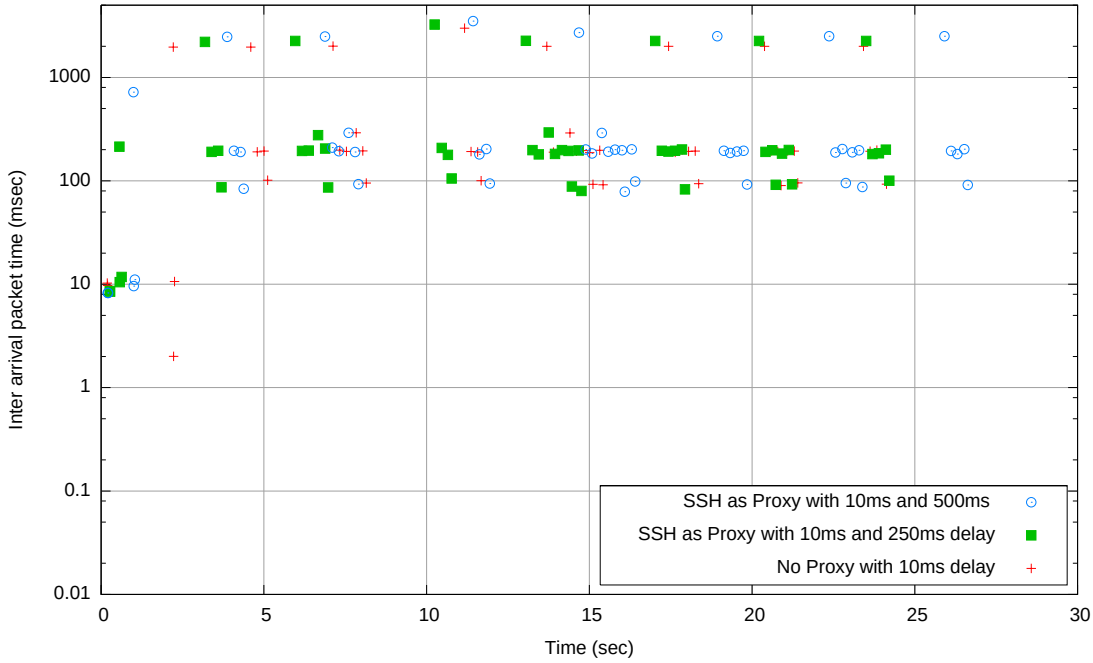

Figure 4.1: SSH test conducted under different network conditions with SSH protocol as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

Analysis and Discussion

In first phase of the experiment, the proxy system is acting as a SSH server for the client C and a SSH client for the server S. The attacker using the proxy P initiates a connection towards the server S. From figure 4.1, we see that during initial time period from 0-1 sec, the packets arrived after approximately 10ms at the server end in all three test cases. These packets belong to the session key setup phase and are triggered by the SSH transport layer [56] protocol. After establishment of the session key, attacker types in a username and a password to log in to the server S. After successful login, the server S opens requested service and the attacker gets the command prompt.

In Figure 4.1 near the period of 5 secs, the first command (pwd) character packet arrives for testcase 4.1. The remaining command packets arrive with IPT value of approx. 200ms, 200ms and 100ms correspondingly. In the test cases [4.2,4.3], the first command character arrives after higher delay compared to the testcase 4.1, but the IPT of all the remaining characters in the command is almost similar to the testcase 4.1. The arrival time of all the remaining command packets shows similar

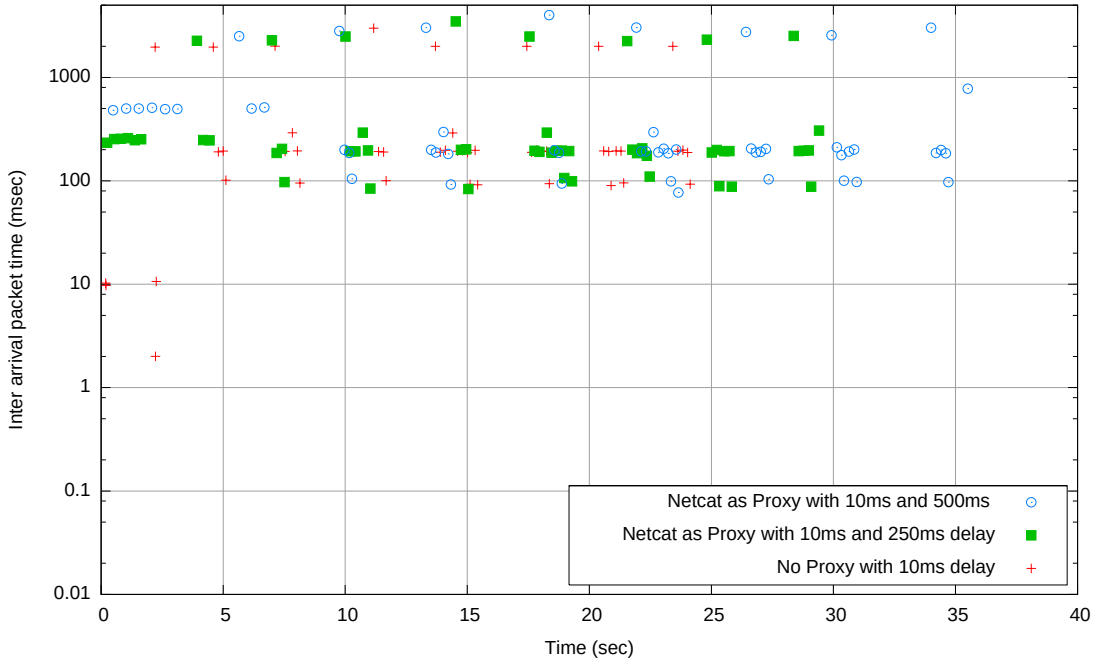

Figure 4.2: SSH test conducted under different network conditions with netcat as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

behavior.

From the above result, we can say that the difference between IPT values from all the three test cases is not significant to make a decision about presence of a proxy system in the connection.

In second phase of the experiment, the netcat is working as a generic TCP proxy and the SSH session is setup between the client C and the server S. We can see that in testcase 4.1, the packets for key exchange arrive after 10ms or slightly higher. When the proxy P is included in the connection, the initial phase packets start to arrive according to delay on the connection between server S and client C. Therefore, the initial packets arrive at the server in testcase 4.2 after 250ms or slightly higher and in the testcase 4.3 after 500ms or slightly higher. Except the change in initial behavior, all the remaining behavior is similar to first phase of the experiment. The IPT values of the command packets result in a similar value in second phase also for all the three test cases.

The reason behind the same values of IPT for command packets is the SSH con-

nection protocol. The SSH connection protocol specifies the $SSH\_MSG\_CHANNEL\_WINDOW\_ADJUST$ (window size) [54] in bytes during channel setup. The window size tells how much bytes other party can send before it must wait to adjust the window. This value of window size is different for the different SSH implementation. For example in $OpenSSH\_5.1p1$ the value is three, after sending three packets the sender waits for a response from receiver before sending more data. This wait can result in a higher delay if the delay on any connection in the connection chain is higher enough to make the distinction noticeable.

4.1.2 SSH Experiment using Script

The motivation for this experiment is that the attackers not always type command characters from their own systems. They can run the script from their system and launch an attack on the victim. The intuition behind this experiment is that when the execution of a command is completed, the result will travel up to an attacker system via chain of stepping-stones. On arrival of the execution result, the attacker will send next command that will travel from the attacker system to victim system via the same stepping-stones. The delay in arrival of the next command will always be equal or higher to total delay on the connection between attacker system and the victim system. Therefore, the difference in delay for arrival of next command is noticeable at the victim side and detection about presence of a proxy is possible in the connection.

This experiment is conducted using the same network conditions and command set as used in section 4.1.1. The script used for the commands has no typing delay and each request packet from the client C carries a complete command instead of a single command character. The experiment is conducted in two phases similar to the section 4.1.1. In first phase the SSH protocol is working as a proxy and in second phase the netcat acts as a generic TCP proxy.

The figure 4.3 shows the result from first phase of the experiment. The Figure illustrates the IPT of incoming request packets from the sender. The next phase of the experiment uses netcat as a proxy and same command script and network conditions to perform the tests. The figure 4.4 illustrates result from this phase of the experiment. The second phase shows the IPT values for key exchange and encrypted packets coming from the sender.

Analysis and Discussion

In first phase of the experiment, the SSH protocol is working as a proxy and a SSH session is established between server S and the peer (P or C). From figure 4.3 we

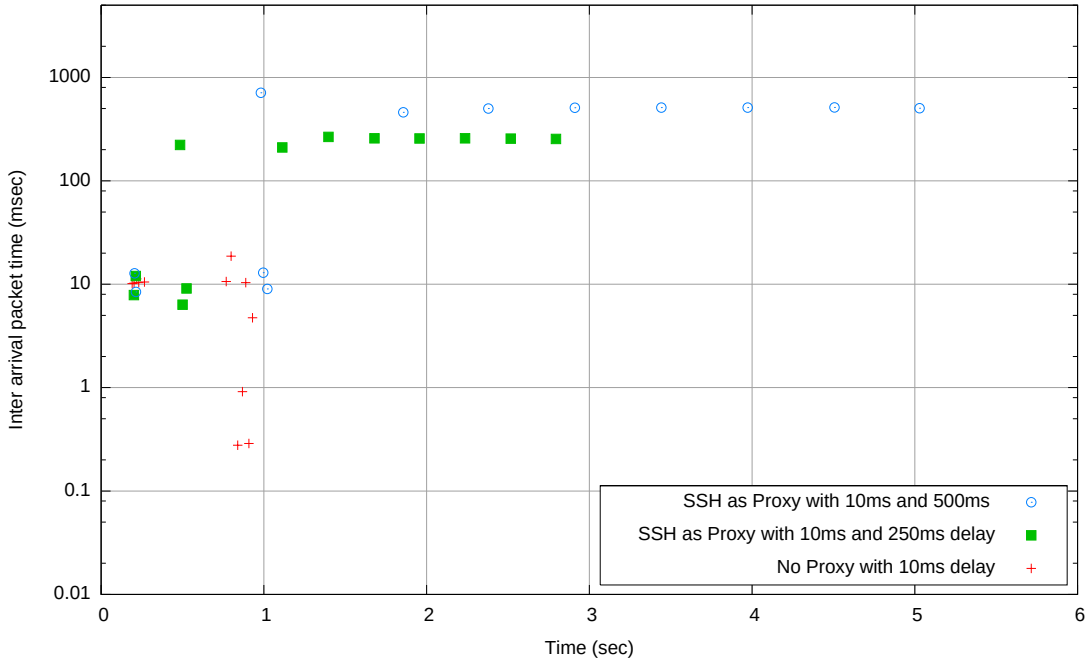

Figure 4.3: SSH test conducted using script under different network conditions with SSH protocol as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

can see that during the initial time period of 0-1 sec, the packets from all the three test cases arrive after 10ms delay or so. After the session key is established, we can see that the username, password and command packets arrive after different IPT values. The IPT values for each packets after the initial setup is corresponding to the delay between server S and client C.

The reason behind this behavior is that the result of last given command travels from the server S to client C and upon receiving the result, the client C sends a next command. The next command arrives at server after covering the complete path from client C to server S. Therefore journey of the result from last command and the next given command results in one complete RTT delay of the connection between server S and client C.

In second phase of the experiment, the netcat is working as a generic TCP proxy and the client C establishes a SSH session directly with server S. This will result in IPT values of initial key exchange packets equal to the delay between server S and client C. After the successful SSH session setup username, password and the command packets arrive with different IPT values at the server side under different

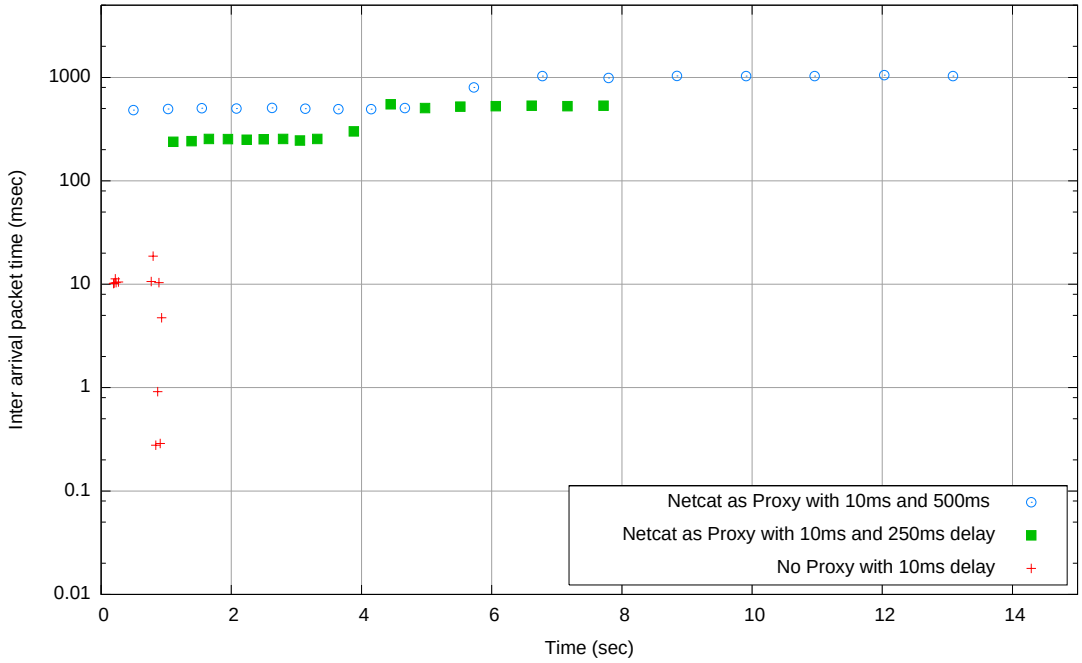

Figure 4.4: SSH test conducted using script under different network conditions with netcat as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

network delay conditions.

In second phase, the username and password at server S arrives after total delay on the connection between server S and client C. This behavior is seen in all the three test cases during the second phase. However, the command packets in proxy case arrive after twice as much as delay on the connection between server S and client C. This behavior is seen in both the test cases 4.2,4.3. The reason behind this behavior is nagle algorithm inhibits the proxy system from sending small size segments to client and makes it to wait for an ACK to arrive from the client. The reason for occurrence of this behavior is the use of netcat as TCP proxy. In the first phase SSH sets the window size bigger and avoids this wait, and result in delay equal or slightly higher than RTT delay on the connection between server S and client C.

From the results obtained in second experiments, we can detect the presence of proxy with confidence by monitoring the IPT value of request packets from the client. The decision about use of script by client can be made by monitoring the number of reply packets sent by server in response to single request packet. Either

to avoid the detection, the attacker can send the command characters one by one without any delay or he can run script from the last intermediary host. In former case, the window size will make client to wait and result in delay of packet arrival at the server S. In later case, the script run from last intermediary host will result in a communication pattern similar to a legitimate case, and therefore detection is not possible.

The conclusion from SSH protocol experiments is that the detection about presence of a proxy system in the incoming connection is possible in two cases. First if an attacker uses generic TCP proxy to relay his attack, then detection is possible in early phase during session establishment. The user typing speed is a main variable and need to choose carefully to detect the proxy. To detect the presence of a proxy system in a connection under OpenSSH, the following equation must satisfy.

$$4 * x < d \tag{4.4}$$

Where x is average typing speed/character (msec) of a user and d is a delay on any connection in the connection chain up to last intermediary host. Secondly if an attacker uses a script to launch an attack, then the script should run from his own system or any other proxy system on the path between attacker system and the second last proxy system. In other cases, it is not possible to detect the presence of proxy system in an incoming connection due to the SSH custom window size.

4.1.3 The Approach Implementation for SSH Protocol

To implement the given approach for detection of a proxy system in an incoming SSH connection, the Bro [41] intrusion detection system is used. The code is written in a script file used by Bro to detect network based attacks. The script can detect presence of a generic TCP proxy system in an incoming SSH connection. The complete source code is given in Appendix A.1

In Bro, the events are generated when a packet arrives at the server. There are different kinds of events e.g. new_packet, connection_established, login_success etc [3]. The script files are used as the policies, which tells Bro to register the analyzer to analyze an event of interest. The round trip time of the every TCP connection is calculated during 3-way handshake and stored in a table with corresponding connectoin ID. This is performed by making a addition in a connection script “conn.bro“ provided with the Bro. The patch is provided for the modifications done in ”conn.bro”.

When a new TCP connection is established with the server, the script checks each newly established connection whether it is either a SSH connection (port 22)

or any other connection. If it is a SSH connection then the script start to monitor IPT values of incoming packets with packet size greater than zero. The value of connection's RTT delay is fetched from the `conn.bro`.One more factor corresponds to Host overload factor is added in to RTT delay. This factor is to take in account a delay caused by work load at the host in responding to a request. The RTT delay condition against which the IPT values are checked is calculated as.

```
session$rtt = connection_rtt(c$id);
if (session$rtt > .05sec)
       session$rtt_cond =  session$rtt*1.25;
else
       session$rtt_cond =  session$rtt*2.5;
```

The scripts monitors some initial incoming packets, which belong to the key exchange and session setup phase. After recording the IPT values of these packets, the average of recorded IPT values is calculated.

```
if(session$avg_t > session$rtt_cond)
       {
       print proxy_file, fmt ("%.6f: #%s Proxy Connection %s", \
                        network_time(), prefixed_id(session$id), \
                        id_string(c$id));
       session$alarmp = T;
       }
```

If the average IPT value is higher than `session$rtt_cond` then a warning is logged in to a file named `proxy.log`. The script is tested on captured files from the experiment using command as.

```
sudo bro -r ssh_10ms_500ms_nwproxy.pcap -r ssh_script_10ms_250ms_
               nwproxy.pcap -r ssh_10_250ms_proxy.pcap -r ssh_10ms
               _noproxy.pcap ssh-proxy
```

The result obbtained from the above test on captured files is logged in to `proxy.log`. The entry logged into `proxy.log` file shows information about the proxy connection. Each line in the logged entry shows the time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and records IP address of the client with port number used to connect at the SSH server.

```
1243445742.177616: #2 Proxy Connection 129.241.208.54/43945
                                        > 129.241.208.186/ssh
1243448437.259864: #3 Proxy Connection 129.241.208.54/45000
                                        > 129.241.208.186/ssh
```

4.2 Telnet Protocol

The Telnet protocol provides access to a command-line interface of a remote host using a virtual terminal connection over the TCP protocol. The telnet is available for all the current available operating systems. Due to lack of security features, the use of telnet has been stopped for remote access. However, some users still use it due to its widespread availability. The telnet is also used in debugging of network services such as SMTP, HTTP or POP3 to issue commands and examine the responses.

4.2.1 Telnet Experiment using Typing

The motivation for telnet experiment is to detect the use of intermediary hosts in an incoming connection at the receiver end. However, the telnet is not much in use for remote access, but its behavior is similar to any custom built protocol by an attacker to interact with the bots or to send commands to the bots in his botnet. The study of telnet behavior under different delay conditions helps in detecting the use of intermediary hosts in their command and control network. Also the lack of security features makes it vulnerable to *man in the middle attack* and using this study it can be possible to detect such attack. The intuition behind the telnet experiment is similar to SSH experiment. In a connection chain, the journey of command characters from the attacker system will result in a higher delay at the victim system compare to delay in arrival of characters from the last intermediary host in the chain.

The telnet experiment is conducted with the three computer systems. The system S is acting as a telnet server, the system P is acting as a proxy system and the system C is acting as a telnet client. The attacker is located at the client system. The proxy system P can act as either a telnet server for client C or telnet client for server S or as a generic TCP proxy using netcat. The command set used in telnet experiments is similar to SSH experiments. The delay in typing command character is kept similar to the SSH experiment. The telnet experiment is conducted in two phases.

In first phase, the proxy system is acting as a telnet server for client C and telnet client for server S. During the first test, there is no proxy on the network path between client C and server S. The delay condition on a connection between client C and server S is as shown below.

$$S \xleftrightarrow{10ms} C \tag{4.5}$$

Now a proxy system is included on the path between server S and client C. The

network conditions on the path are as

$$S \overset{10ms}{\longleftrightarrow} P \overset{250ms}{\longleftrightarrow} C \tag{4.6}$$

In third test, the delay on a connection between proxy P and client C is increased up to 500ms.

$$S \overset{10ms}{\longleftrightarrow} P \overset{500ms}{\longleftrightarrow} C \tag{4.7}$$

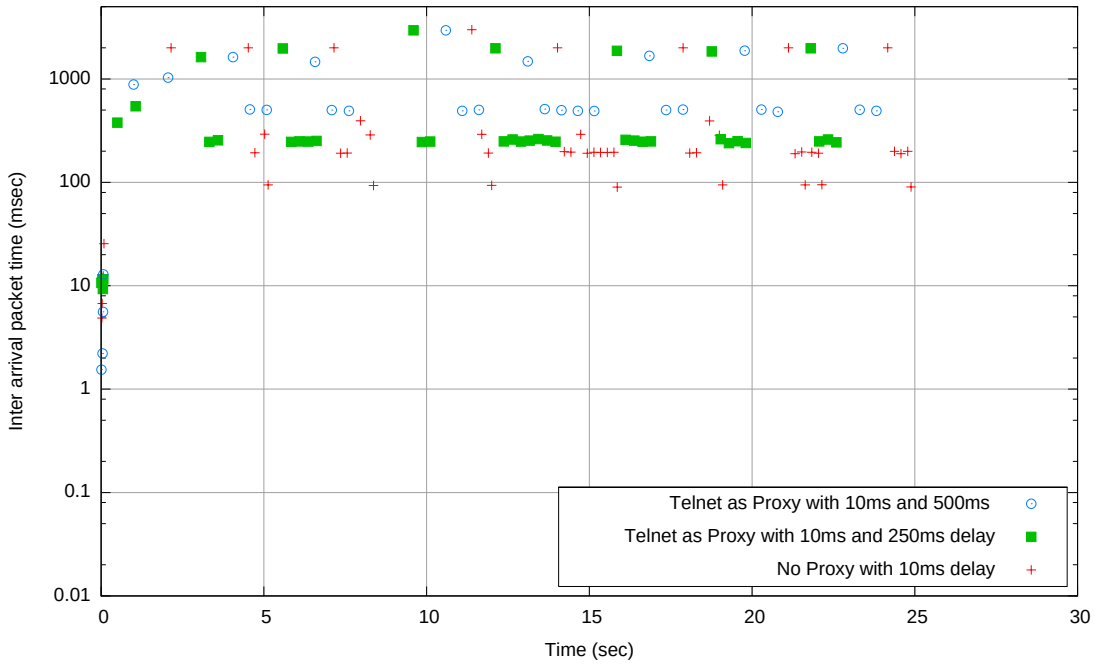

Figure 4.5: Telnet test conducted under different network conditions with telnet protocol as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

The Figure 4.5 illustrates result from first phase of the experiment. The figure shows the IPT of incoming request packets from the peer system (P or C) with size greater than zero. These packets belong to telnet session negotiation, username, password and command characters.

In second phase of the experiment, the proxy system P is acting as a generic TCP proxy between client C and server S. The telnet session is setup between client C and server S. The network delay conditions and command set is similar to the first phase. The script used in the test to type command characters has the similar

delay as in the first phase. The result from second phase of the experiment is shown in Figure 4.6. The figure illustrates the packets from the peer system (P or C) with size greater than zero.

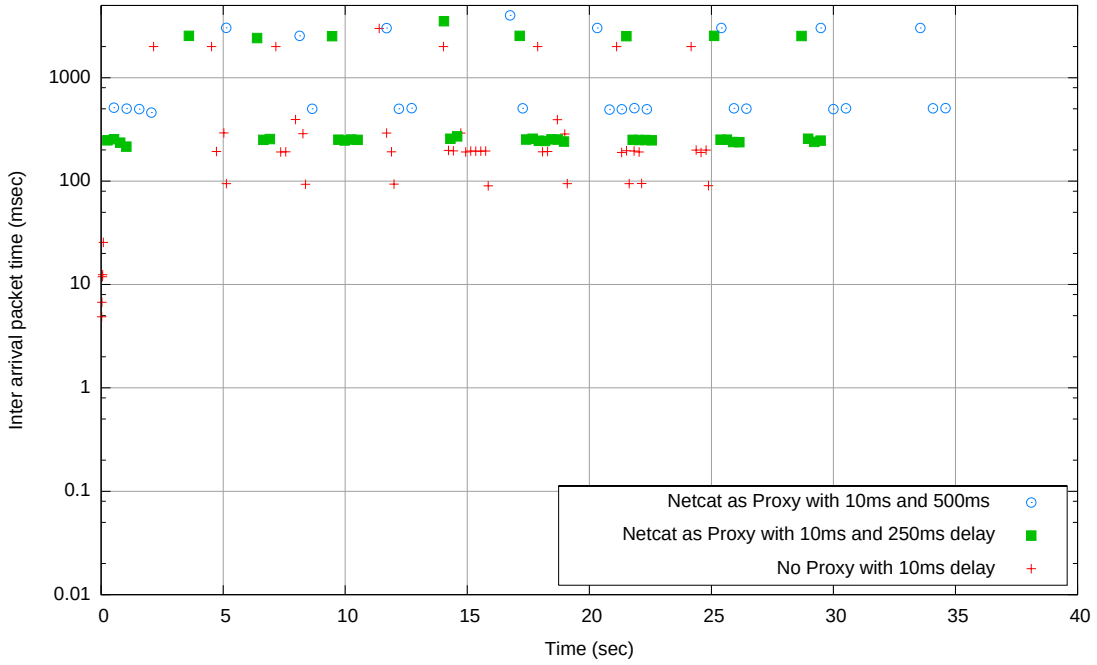

Figure 4.6: Telnet test conducted under different network conditions with netcat as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

Analysis and Discussion

The result from telnet protocol experiment shows that the IPT values of incoming request packets is distinguishable in all three different test cases. In first phase of the experiment, the telnet session is setup between the server S and the peer system (P or C). The packets belong to session negotiation arrived at the sender around same time in all three test cases. From figure 4.5, we see that the packets belong to command characters arrived at different IPT values in all three test cases considering the delay in typing the characters is same. We see that the command packets in testcase 4.6 always arrived after delay of 250ms or more. The same behavior is seen in the testcase 4.7, where packets arrived after 500ms or higher.

In second phase of the experiment, the session is setup between the client C and the server S. This results in arrival of the session negotiation packets after total delay

on the connection between client C and server S. Therefore monitoring IPT values of the incoming packets and comparing it with delay on the connection between server S and the peer communicating with it (P or C), a decision about presence of proxy can be made. When a TCP generic proxy is used the decision about proxy presence can be made in earlier phase compared to the case when telnet protocol is used as a proxy. The command packets arrived in a similar behavior as seen in the first phase and result in distinct behavior under different test cases.

The reason behind the distinct values of command packets in different test cases is that in telnet protocol, the client waits for a reply to come from the server before sending another data packet. In telnet protocol the size of request packet is small(1 byte/character) and the Nagle algorithm inhibits client system from sending next packet before receiving an ACK from the peer (P or S). The client system will buffer the command characters and send them together upon arrival of ACK packet. The wait in sending data at the client side results in delay equal to an RTT delay on the connection chain up to the server. From figures 4.5 and 4.6, we can see that packet numbers are also less in test case 4.7 for same command compared to other test cases which signifies the buffering of command character at the client side.

4.2.2 Telnet Experiment using Script

The telnet experiment using script is performed with a motivation to study the IPT behavior in case of script used by an attacker to launch an attack. The intuition behind this test is similar to section 4.1.2. The next command from attacker will arrive after total RTT delay on the connection between an attacker and the victim system.

The experiment is conducted using the network delay conditions and command set similar to experiment in section 4.2.1. The script used to send commands has no typing delay and each request packet from the attacker carries complete command in spite of single command character. The experiment is conducted in two phases similar to the first experiment under telnet protocol.

In first phase telnet protocol is acting as a proxy. The result from first phase of the experiment is shown in figure 4.7. The figure illustrates the IPT of incoming request packets from the peer (P or C) with packet size greater than zero.

In second phase of the experiment, the netcat is used as a TCP generic proxy. The result from second phase of experiment is shown in figure 4.8. The figure shows IPT of the incoming request packets from the peer (P or C) in all three different

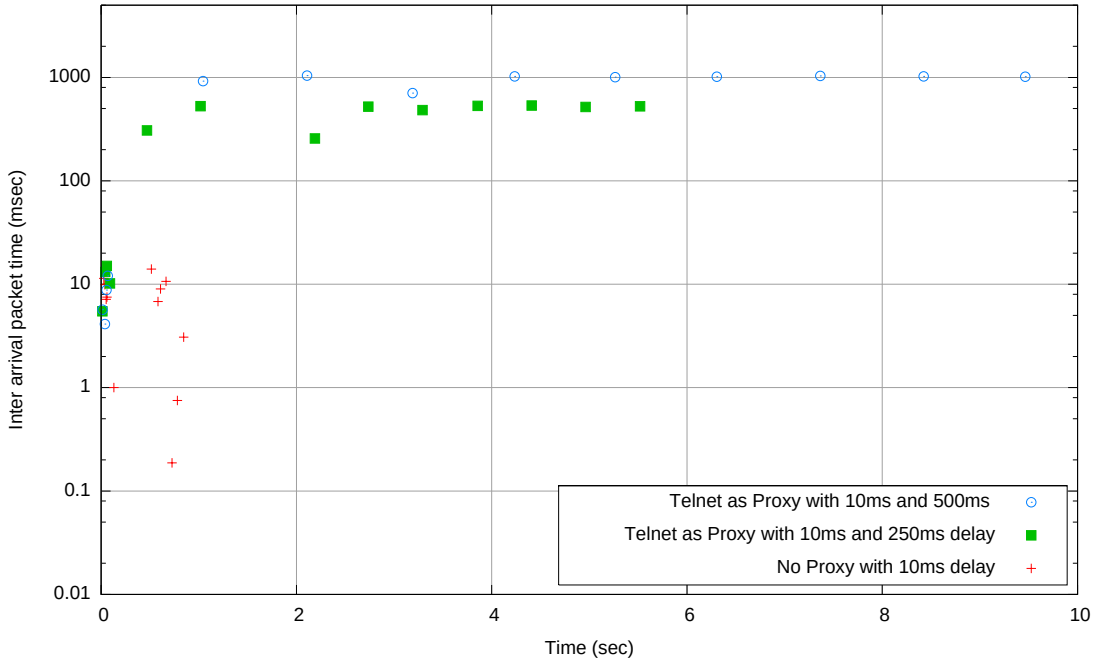

Figure 4.7: Telnet test conducted using script under different network conditions with telnet protocol as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

network conditions. The packets belong to telnet session negotiation, username, password and commands.

Analysis and Discussion

In first phase of the experiment, the Telnet protocol works as a proxy and session is setup between the proxy P and the server S in test case 4.6 and 4.7. From figure 4.7 we can observe that in the initial period of 0-1 sec, the session setup packets arrive with approximately 10ms delay in all the three test cases. After the session is established, packets from the peer (P or C) arrive after the corresponding delay on the connection between server S and client C in all the three test cases. The IPT values of incoming packets from the peer is either equals or higher than the delay on connection between server S and client C.

The explanation for the shown behavior is the result from last given command travels from the server S to client C and upon receiving the result, client C sends a next command. The next command arrives at the server after travelling complete

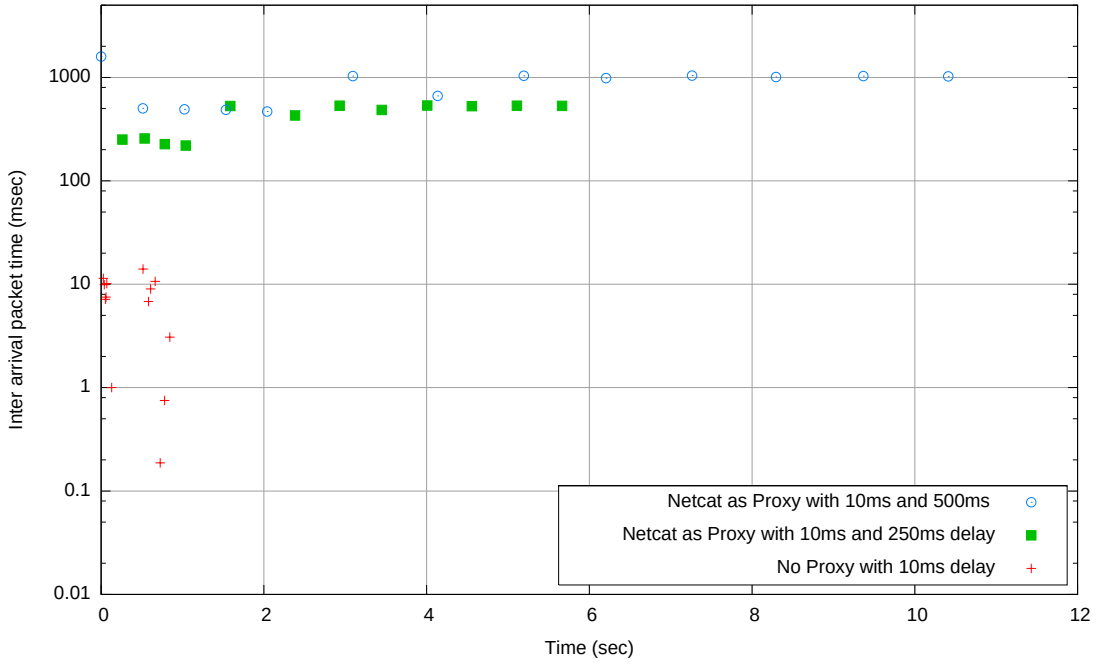

Figure 4.8: Telnet test conducted using script under different network conditions with netcat as a proxy. The X-axis shows Time (sec) and the Y-axis shows the inter arrival packet time (msec) on log scale.

path from client to server via proxies in the path and results in delay equal or higher to the RTT on the connection between server S and client C. From figure 4.7, we see that command packets in test case 4.6 and 4.7 arrives after almost two times the delay on the connection between server S and client C. The reason is the nagle algorithm inhibits proxy system from sending small size reply packets form server back to the client and waits for an ACK to arrive from client. This wait couples with delay in arrival of the next command from client at proxy and results in almost twice as much as RTT delay on the connection.

In second phase of the experiment, the netcat acts as a TCP generic proxy and telnet session is established between the server S and the client C. From figure 4.8, we can see that the packets from session negotiation arrive with different IPT values in all three test cases. Except this difference, the result in second phase is similar to first phase of the experiment.

The conclusion from telnet protocol test is that an incoming connection can be distinguished from a connection coming via some proxies or a connection coming directly from sender at the receiver end. The nagle algorithm couples with allowed

window size results in a wait at the sender side. The wait causes a delay equal to maximum delay on any connection in the connection chain. The user average typing speed per character must satisfy the following equation for detection to happen when telnet protocol is used as a proxy.

$$2 * x < d \tag{4.8}$$

Where x is average typing speed/character and d is delay on any connection in the connection chain up to last proxy system.

4.2.3 The Approach Implementation for Telnet Protocol

The implementation of proposed approach is performed in a script file named "telnet-proxy.bro". This file can be loaded in to the Bro to start monitoring the incoming Telnet connections. The complete source code of telnet script is given in Appendix A.2. The script is monitoring the `event connection_established` and checks whether the newly established connection is a telnet or any other connection. If it is a telnet connection then the script makes a new copy of session record to store the information related to this session. The information for all the current sessions is store in a table named "telnet_sessions".

The Telnet script performs detection of a proxy system in the two phases. In first phase, the script tries to detect the presence of a generic TCP proxy in an incoming Telnet connection. This phase monitors 4 initial incoming packets and compares the average of IPT value against `session$rtt_cond`. If the average is higher than the rtt condition then a warning is raised. The reason for choosing only 4 initial packet is based on number of packets used by the Telnet to negotiate the session options. If the session is setup between the server and the peer (Proxy or client system), then the detection from this phase is not possible.

If the first phase is not able to detect the proxy system then second phase starts working after the login is successful. In second phase the `session$rtt_cond` is calculated as.

```
session$rtt = connection_rtt(c$id);
if (session$rtt > .05sec)
       session$rtt_cond =  type_speed + (session$rtt*1.25);
else
       session$rtt_cond =  type_speed + (session$rtt*2);
```

The `type_speed` varaiable specifies the user's avg. typing speed (msec) per character. This variable can be tuned to optimize the detection process. During the

experiments performed, the value of 250 msec is found adequate. The script monitors the incoming connection either for first 10 command packets or for a duration of 90 sec in case of number of incoming command packets are less. The IPT value of incoming packets is stored and the average is taken after one of the mentioned condition is satisfied. If the average found higher than the `session$rtt_cond`, then a warning is logged in to the file named *`proxy.log`*. To test the script, it is run on the captured files using Bro as follows.

```
sudo bro -r telnet_10_500ms_nwproxy.pcap -r telnet_10ms_noproxy.pcap
        -r telnet_10_250ms_nwproxy.pcap -r telnet_10_250ms_proxy.pcap
        -r telnet_10ms_500ms_proxy.pcap telnet-proxy
```

After running the script on these captured files, the warning is logged in to `proxy.log`. Each line in the logged entry shows time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and logs IP address of the client with port number used to connect at the Telnet server.

```
1243446985.816970: #1 Proxy Connection using generic TCP proxy
                        129.241.208.54/55828 > 129.241.208.186/telnet
1243447096.503351: #2 Proxy Connection using generic TCP proxy
                        129.241.208.54/55831 > 129.241.208.186/telnet
1243446734.365517: #3 Proxy Connection 129.241.208.54/54042
                                        > 129.241.208.186/telnet
1243447927.985351: #4 Proxy Connection 129.241.208.54/49747
                                        > 129.241.208.186/telnet
```

4.3 Application to Manual Intrusion Detection

The possible application for the experiments performed to study remote login protocols is the *Manual intrusion detection*. The attackers generally launch their attacks after logging in to a chain of stepping stones to avoid the possibility of being traced and get punished. If during an active communication session, the presence of a proxy system can be detected then a warning will be raised and according to organization policies an administrator can either closely monitor the connection or terminate it.

In remote login protocols, the main variable in decision about pthe resence of a proxy system is the user's average typing speed per character. The average typing speed of a user can cause high false positive rate if sets too small and high false negative rate if sets too high.

"Vision without action is empty. Action without vision is blind."

Yasuhiko Kimura

5 The FTP and HTTP Protocols

5.1 FTP Protocol

The File Transfer Protocol (FTP) is designed to provide file sharing among the hosts on a network. The FTP protocol hides the underlying file systems variations among the hosts and provides a reliable and efficient data transfer among them. The FTP protocol uses two different connections to provide the required functionality. The FTP client initiates a connection with the FTP server on port 21, where server listens for the commands from a client and executes them. After the execution of a given command, the result is transferred to the FTP client using FTP-data port (default 20) of the server.

5.1.1 FTP-Control Protocol Experiment

The motivation for a FTP experiment is to detect the use of any proxy system in the incoming FTP connection at the server. The attacker can use the proxy system to setup a FTP connection with the server and can upload any malicious file. Moreover, the FTP is one of the widely used protocols to transfer files among the computers on a network. Therefore, it is worth to study its behavior in presence of a proxy system in the path and check the possibility of detection.

In this experiment, four computer systems are used under the different network conditions. The command set consists of five commands. The commands are "ls, cd *dirname*, ls, put *filename* and exit". During the experiment, a file is uploaded at the FTP server and complete session is captured in a trace file at the server side. At the proxy system, a FTP proxy (ftp.proxy tool) is running to relay the connection from a client to a server. In first test case, a direct FTP connection is setup between the FTP client and the FTP server with 10 ms delay on the connection.

$$S \xleftrightarrow{10ms} C \tag{5.1}$$

After the successful login above mentioned command set is run in corresponding order and packets are captured for the session at the server end. After this, a proxy system is introduced under network delay conditions as.

$$S \overset{10ms}{\longleftrightarrow} P \overset{250ms}{\longleftrightarrow} C \tag{5.2}$$

The client C makes a request to proxy P and the proxy forwards connection request to the server S. After the successful login, above mentioned commands are run and packets are captured at the server side. The same experiment is done under different delay conditions as

$$S \overset{10ms}{\longleftrightarrow} P \overset{500ms}{\longleftrightarrow} C \tag{5.3}$$

Now one more proxy system is introduced in the connection chain with FTP proxy running on it. The delay on connection chain is as shown below.

$$S \overset{10ms}{\longleftrightarrow} P_2 \overset{300ms}{\longleftrightarrow} P_1 \overset{250ms}{\longleftrightarrow} C \tag{5.4}$$

After successful login to the server S, the same command set is run and packets are

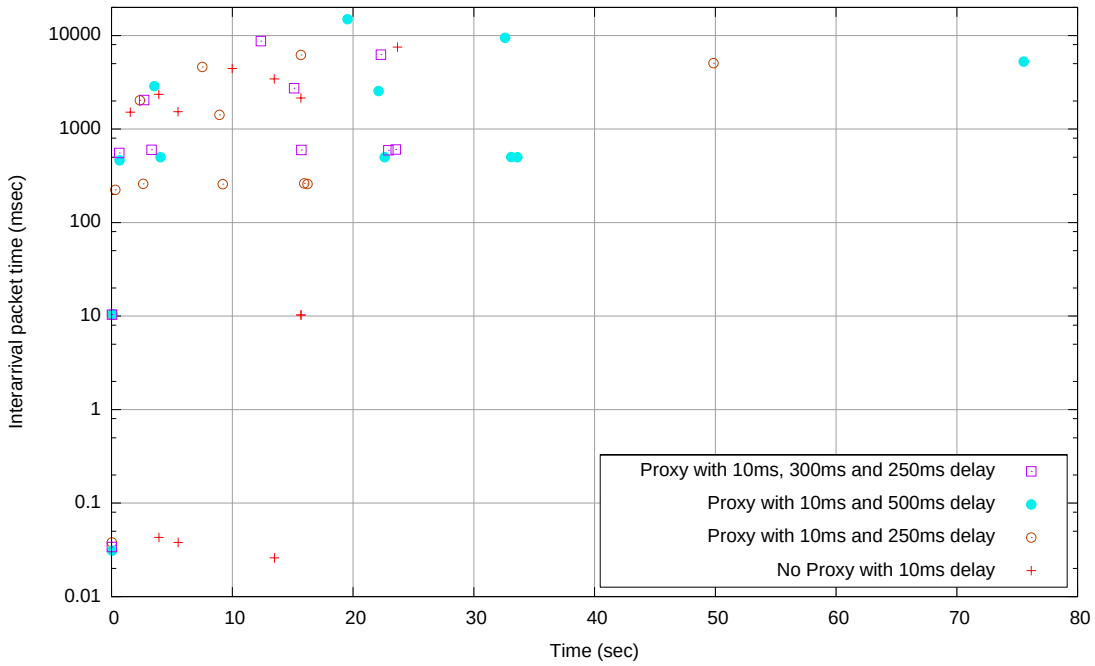

Figure 5.1: FTP-Control connection test result. The X-axis as time (sec) and the Y-axis as interarrival time (msec) of ftp command packets on log scale.

captured at the server side. The main emphasis is on the interarrival packet time of

FTP command packets. The Figure 5.1 illustrates result of interarrival packet time of the FTP command packets under the different test conditions. In Figure 5.1, the result marked as "Noproxy 10ms delay" shows result from the test case 5.1 and result marked as "Proxy with 10 ms and 250ms delay" shows result from the test case 5.2. The similar notation terminology is used in the tests performed in this experiment.

Analysis and Discussion

From the figure 5.1, we see the distinct behavior at the server side under different network conditions. The initial phase in all the test cases results in almost same IPT values of the incoming packets. These packets belong to the session setup and user authentication phase. The reason for the similar value of IPT is the use of an application level proxy for the FTP client.

After successful authentication, the attacker starts to send commands. In FTP protocol, the commands are not sent character by character as seen in SSH or Telnet. In FTP, the complete command arrives at the server S, after the attacker hit enter key at the client C. In FTP protocol, a different connection is setup between the server S and the peer (P or C) to transfer the response data of a given command. To establish a new connection to transfer data, the client C sends a port number on which it will listen for the incoming connection request from the server S.

Consider an example of first command "ls" in which the server has to transfer a list of directories and files in the present directory. After receiving the command, the FTP client sends a PORT [44] request to the server S with a given port number on which it is listening for the data connection. The server S sends a response suggesting that the PORT command is successful. The given response travels the complete path from the server S to the client C via, if any, proxies present in the network path. The client upon receiving successful response (code 200) sends the actual command LIST (in given case). Therefore, the LIST command arrives at the server after the total delay of connection between the server S and the client C and results in a noticeable difference under different network conditions.

In the test case 5.1, we see that in "ls" command case, PORT request arrives after 1.5 sec and then the command LIST arrive after .038 msec. The reason is that the command LIST arrives with an ACK packet after 10ms delay, which is a delay between the server S and the client C in test case 5.1. Consider the test case 5.3, we see that the PORT request packet arrives after 2.8 sec and the command LIST arrives after 500msec delay, which is a delay between the proxy P and the client C. The similar behavior is seen in all other command packets.

In the test case 5.4, there are two proxy systems are included in the path. After successful execution of the PORT command, the LIST command arrives after 590ms which is slightly higher than the delay between server S and client C. The slightly higher value of IPT than the total delay of connection between the server S and the client C is due to the additional delay caused by the system load. Thus monitoring IPT values of the FTP command requests, a decision can be made about presence of a proxy system in the incoming FTP connection at the receiving end. The ACK packet arrives at the server side after the delay on connection between the server S and the peer (P or C). This signifies the lack of congestion on the network path.

The conclusion from the FTP protocol experiment is that an incoming direct connection is distinguishable from a connection coming via a proxy system. The distinction is possible due to the setup of FTP-data connection to transfer data in FTP protocol. In FTP protocol, the average user typing speed/character is no longer a problem. Because the command arrives at the server after user presses the enter key. If a generic TCP proxy is used then detection is possible in the session setup phase and the FTP command packets show similar behavior as seen in the application level proxy case.

5.1.2 The Approach Implementation for FTP-Control Protocol

In FTP-control connection, the implementation code is written in a script file named `"ftpcon trol-proxy.bro"`. The script can be loaded in to Bro IDS to monitor the incoming FTP connections. The complete source code of the script is given in Appendix A.3.

The script uses another script named “ftp.bro“ provided with Bro IDS. In ftp.bro, a session is created when the server receives a FTP request. To implement the given approach some modifications are made in the ftp.bro and a patch is provided for all the modifications made in ftp.bro.

The script monitors the `event ftp_request` and records the IPT value of incoming request packets. The IPT value of incoming request is stored only when it is below a threshold (`rtt_threshold`) value.

```
if ((session$request_t - session$last_request_t) > rtt_cond &&
                        (session$request_t - session$last_request_t)
                        < rtt_threshold)
       session$sum = session$sum + (session$request_t -
```

```
session$last_request_t);
```

This is used to avoid the high delay caused by user in sending each command to the server. The IPT value of the first four command packets satisfying above conditions is stored in a variable (`session$sum`). The average of these four values is taken and if the average is higher than the rtt condition then a warning is logged in to a file named *proxy.log*. The script is run on the traffic files captured from the FTP experiment. The Bro IDS is used for detection of a proxy system in the connection.

```
sudo bro -r ftp_proxy_10_500msdelay.pcap -r ftp_2proxy_10_300ms_
           250msdelay.pcap -r ftp_noproxy_10msdelay.pcap -r
           ftp_proxy_10_250msdelay_loss 10%.pcap mt ftpcontrol-proxy
```

After running the script on traffic files, the warning for a proxy system in the connection is logged in to the proxy.log file. Each line in the logged entry shows time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and records IP address of the client with port number used to connect at the FTP server.

```
1239978506.022739 #1 Proxy Connection: 129.241.209.164/59259
                                        > 129.241.208.198/ftp
1239980048.518688 #2 Proxy Connection: 129.241.209.164/40635
                                        > 129.241.208.198/ftp
1239980702.719545 #4 Proxy Connection: 129.241.209.164/54710
                                        > 129.241.208.198/ftp
```

5.1.3 FTP-Data Protocol Experiment

The motivation for FTP-Data connection experiment is to detect the presence of a proxy system in the incoming data at the receiving end. The motivation is that the attacker can use some proxy system to relay the malicious data to the FTP server. Therefore, if possible, the detection of a proxy system in this case will result in a warning at the server side, and the administrator can either monitor the connection closely or terminate it.

The experiment is conducted with the three computer systems under different network conditions. During the test, a file of 4.1 MB size is transferred from the FTP client to the FTP server. In the first test case, a direct connection is setup between the server S and the client C with connection delay of 10ms.

$$S \xleftrightarrow{10ms} C \tag{5.5}$$

In second test case, a proxy system is included and the connection chain is as shown below.

$$S \overset{10ms}{\longleftrightarrow} P \overset{500ms}{\longleftrightarrow} C \tag{5.6}$$

In third test case, the delay between client C and proxy P is kept same to 500ms. In this test, a 10% packet loss rate is induced to study change in behavior at the receiving end.

$$S \overset{10ms}{\longleftrightarrow} P \overset{500ms(10\%loss)}{\longleftrightarrow} C \tag{5.7}$$

The Figure 5.2 illustrates result from the FTP-data experiment. The result shows IPT values of the data packets transferred on a FTP-data connection. The notation used in the tests is similar to the FTP-connection experiment.

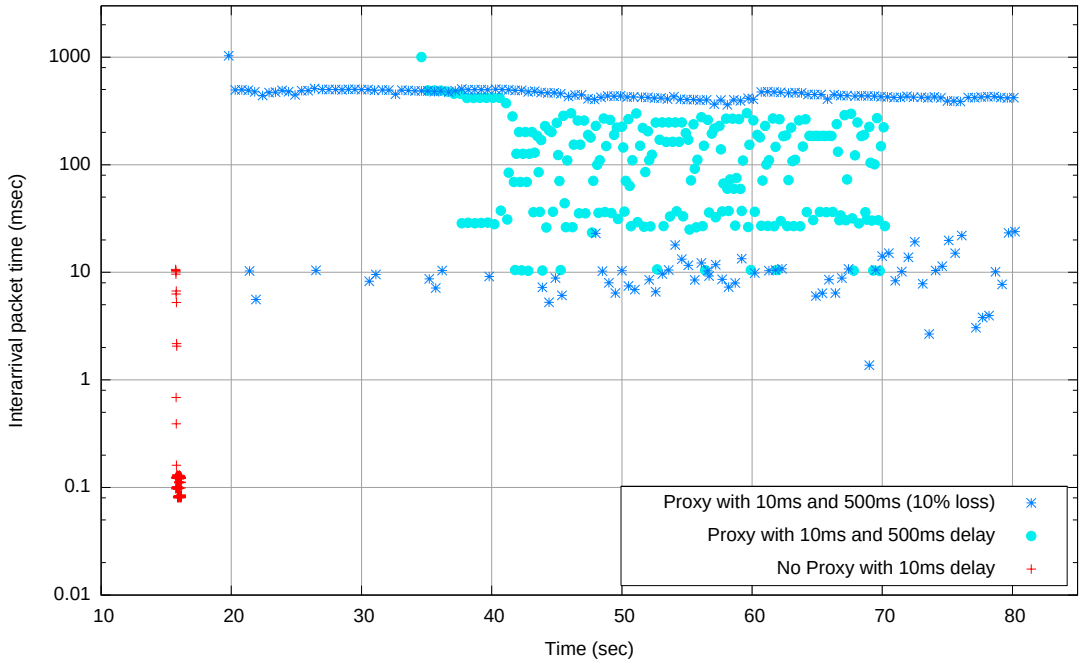

Figure 5.2: FTP-Data connection test result with X-axis as Time (sec) and the Y-axis as interarrival packet time (msec) of data packets on log scale.

Analysis and Discussion

In the test 5.5, we see that majority of the packets arrive with Interarrival Packet Time of 10ms or less. When the data transfer starts, the first packet arrives at the receiver end after two times the total delay on the connection chain. The reason for the noted behavior is that after receiving the STOR request from the client C,

the server S initiates a connection and sends a OK (code 150) response. The response travels up to the FTP client C and upon receiving the response, the client C starts the data transfer. Thus, acknowledgment of the data connection setup and the server OK response causes two times the delay of connection between the server S and the client C.

In the test case 5.6, the packets start to arrive after 500ms delay even though a delay between the server S and the peer (proxy P) is 10ms. The delay value of 500ms is noted during the initial phase of data transfer. The reason for a high delay value is due to the wait for arrival of ACK packet at the client C from proxy P. However, after some time the delay value starts to decrease. The reason is the increase in TCP sender window size, which allows the client C to wait for less time before sending more data. The behavior is similar to the TCP data transfer mode experiment 3.1.2.

An interesting behavior is seen in the test case 5.7, the packet loss is induced in the connection between the proxy P and the client C with probability of 10%. The delay conditions are same as in test case 5.6 and we see the packet start to arrive with delay equal to 500ms. In this test case, the value of IPT remains near to 500ms during the whole data transfer session. The reason for this is the packet loss induced on the connection. The TCP considers the packet loss as a sign of congestion in the network and reduces the window size. The small window size result in a high delay as seen in the given result.

The result from FTP-data experiment shows that if the source is behind another host and relying its data through it, then observing the IPT values, we can identify whether a peer is a original source of the incoming data or just relaying the data.

5.1.4 The Approach Implementation for FTP-Data Protocol

The approach is implemented in a script file named "ftpdata-proxy.bro". The script file can be loaded in to Bro to monitor the incoming FTP-data packets. The complete source code of FTP-data script is given in the Appendix A.4. The script monitors `event connection_established` and if the connection includes FTP-data port then a new copy of session record is made to store information related to the session.

The script fetches the connection round trip time from `conn.bro` and adds a host delay factor based on RTT delay of the connection. The script records IPT value of the packets whose value is higher than the `session$rtt_cond`.

```
if((session$pkt_t - session$last_pkt_t) > session$rtt_cond )
```

```
{
session$sum = session$sum + (session$pkt_t - session$last_pkt_t);
```

After monitoring such five packets, the average of recorded `session$sum` is taken and compared with the `session$rtt_cond`. If the average found to be higher than the `session$rtt_cond`, then a warning is logged in to the *proxy.log* file. The script is run on captured files from the experiment and loaded in to the Bro IDS to detect the proxy connection. The command used to run the script is as.

```
sudo bro -r ftp_proxy_10_500msdelay.pcap -r ftp_2proxy_10_300ms_
           250msdelay.pcap -r ftp_noproxy_10msdelay.pcap -r
           ftp_proxy_10_250msdelay_loss\ 10%.pcap mt ftpdata-proxy
```

The result obtained after running the script on data files is logged in to the proxy.log file. Each line in the logged entry shows time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and records IP address of the client with port number used to connect at the FTP server to transfer data.

```
1239978539.096245: #3 Proxy Connection 129.241.208.198/ftp-data
                                          > 129.241.209.164/52228
1239980063.970564: #6 Proxy Connection 129.241.208.198/ftp-data
                                          > 129.241.209.164/43516
1239980731.406602: #12 Proxy Connection 129.241.208.198/ftp-data
                                          > 129.241.209.164/54805
```

5.1.5 Application to Intrusion Detection

The result from FTP protocol experiments shows that the detection of a proxy system is possible in an incoming FTP connection. The main usage for the proposed approach is the Manual intrusion detection in the case of FTP-control connection. Whereas in FTP-data connection, the main usage is to monitor the incoming data and checks the presence of a proxy system in the FTP connection.

If a proxy system is present in the incoming connection then most likely the incoming file is either a trojan or a malware itself. In both the experiments, if a proxy system is detected then the administrator can take action according to the organization policies. One recent case of manual intrusion in which FTP protocol used is the hacking of `Astalavista.com` server[1]. The hackers use FTP to get contents of the server and later delete most of the contents. If the above approach would have been used in intrusion detection system, then a warning has been raised for the suspicious FTP connection and loss can be avoided.

[1] http://pastebin.com/m4cf55a72

5.2 HTTP Protocol

The Hyper Text Transfer Protocol is an application layer protocol used to transfer virtually all types of file between a web client (browser) and a web server on the network. The HTTP protocol uses TCP as transport protocol to communicate with the peers. The web browser sends a HTTP request to the web server and the server reply with HTTP response, which includes result of the received HTTP request. The HTTP protocol is the main protocol used for accessing email, writing blogs, downloading any file or access any information resource etc on the network.

5.2.1 Tor Protocol

The Onion Router (Tor) [48] is the network of virtual tunnels, which provide anonymous communication to improve privacy of its users on the internet. The anonymous communication network is first introduced by David Chaum in [23], describing the use of mix as a fundamental building block for anonymity. A mix act as a store and forward relay that hides the communication correspondence between messages it receives and forwards. Several architectures are proposed based on the mix architecture and implemented to provide anonymize email, most notably Babel [30], and Mixminion [24]. The problem with the mentioned architectures is their high latency and is not suitable for web browsing. The Tor is a low latency network and work on stream level and support high-bandwidth anonymous communication [46].

The Tor architecture is similar to conventional circuit switched networks. The connection is setup carefully with randomly selected Tor node to preserve anonymity. The connection is established by negotiating self-signed ephermal Diffie Hellman key exchange [25]. The Transport Layer Security (TLS) is used to provide forward secrecy and data integrity. In Tor, the initiator connects to three Tor nodes (default settings) which relay traffic to the destination. The Tor does not perform explicit mixing and data cells are store in separate buffers for each stream connection. The data cells are forwarded from Tor nodes in a round robin fashion and if any buffer is empty, it is skipped and data from the next buffer is forwarded. This provides low latency in the communication and avoids unnecessary delay.

The Tor is used by people from all lifestyles from journalists, militaries to common people to hide their identity on the internet. The problem with Tor is its mis-use by malicious users. From study conducted in [38], the authors suggested that Tor is mis-used mainly for copyright infringement, hacking attempts and web page defacement. In [38], authors suggested a method to detect the malicious Tor exit node is whether logging the initiator data or not. However, there is not much research is performed to detect the request arriving at the HTTP server is coming

via a proxy such as a Tor framework or not. If such detection is possible then it will help in fighting against copyright infringement and avoiding innocent users from being punished due to mis-use of their systems by the hackers.

5.2.2 The HTTP Experiment

The HTTP experiment is performed with a motivation to detect the presence of a proxy system in an incoming HTTP request at the web server. The objective for the experiment is to detect the use of Tor framework at the web server due to its misused in copyright infringement or web page defacement etc. The intuition behind the detection process is that a web page contains a lot of rich contents and in the HTTP protocol to fetch a new resource a new GET request is made from the web client. This GET request will travel from the actual web client system in connection chain up to the web server and thus will arrive at the web server after total delay of the connection chain. Generally a new GET request preceded by setting up a new TCP connection. Thus, the request of new TCP connection (SYN packet) will also arrives after total delay on the connection chain.

The experiment is performed with the three computer systems. The Apache web server is setup on system S, a system P is acting as a proxy system and a system C is acting as a web client. A webpage with rich content is hosted at the web server and test is performed under the different network conditions. In starting, the test is performed without any proxy in the connection between the server S and the client C with network conditions as.

$$S \xleftrightarrow{20ms} C \tag{5.8}$$

Now a generic TCP proxy system is included between the web client C and the web server S under network conditions as.

$$S \xleftrightarrow{20} P \xleftrightarrow{110ms} C \tag{5.9}$$

Now to study behavior of the anonymous proxies available on the internet, a test is conducted using one of the available free open proxies. The proxy server used during the test is *www.killmyip.com*. The delay between the proxy system P and the web client C and a delay between the proxy system P and the web server S is as shown below.

$$S \xleftrightarrow{135ms} P \xleftrightarrow{135ms} C \tag{5.10}$$

Now in another test the behavior of Tor framework is studied at the web server side for an incoming request in a HTTP session. In the Tor framework, the client establishes a secure connection among three machines in a connection chain up to the server S. The test is performed with conditions as shown below.

$$S \xleftrightarrow{50ms} P_3 \xleftrightarrow{unknown} P_2 \xleftrightarrow{unknown} P_1 \xleftrightarrow{185ms} C \tag{5.11}$$

The Figure 5.3 illustrates result from the HTTP experiment performed under

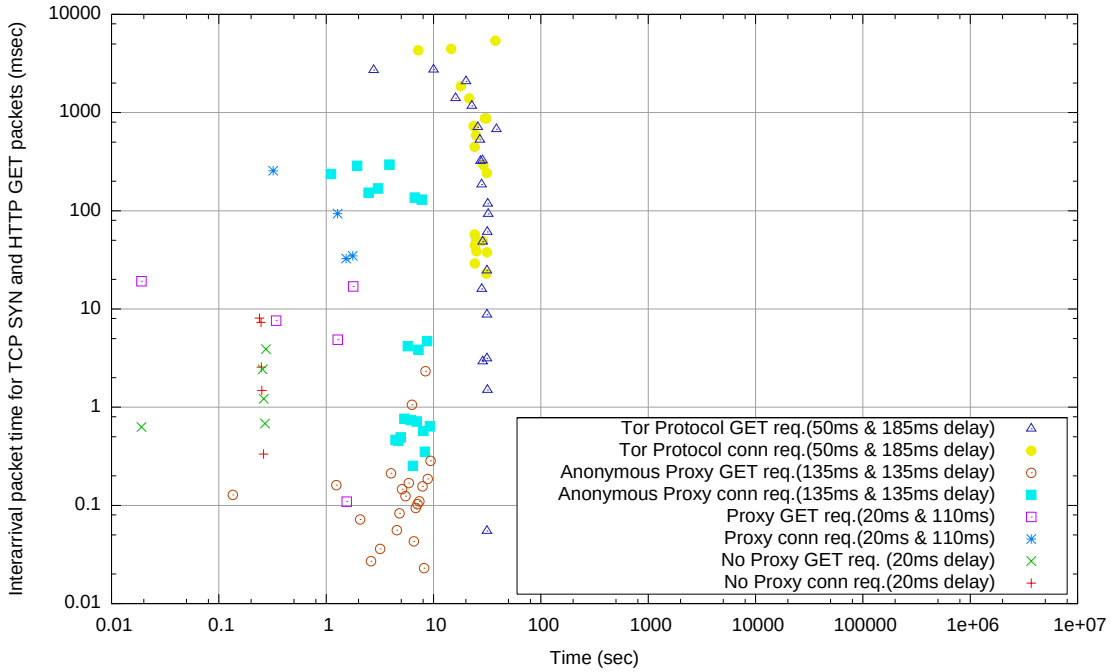

Figure 5.3: New TCP connection (SYN packet) and HTTP GET request interarrival time during HTTP session test. The X-axis as Time (sec) on log scale and the Y-axis as interarrival packet time (msec) of SYN and GET request packets on log scale.

different network conditions. The result shows Interarrival Packet Time of a new connection request (SYN packet) and a new GET request in a HTTP session. The result labeled as "No proxy conn req.(20ms delay)" shows the result of IPT values from SYN request packets in the test case 5.8. The result marked as "No Proxy GET req(20ms delay)" shows the result of IPT values from GET request made on the newly established connection in the test case 5.8. The result marked as "Anonymous proxy conn. req.(135ms & 135ms delay)" shows the result of IPT values of SYN request packets under the test case 5.10 and similar for GET request too. The similar notations are used for other test cases.

Analysis and Discussion

The figure 5.3 shows IPT values of the incoming request packets under the different test cases. In the test case 5.8, the new connection request arrives with in 20ms delay and the GET request on a newly established connection arrives with in 20ms, which is a delay between the web server and the web client. In the test case 5.9, a

generic TCP proxy is used and proxy tool allows web client to establish connection on more than one port as required in HTTP session to fetch new web resources. The result shows that the first two new connection request arrives after 100ms delay or higher, but later the new connection request arrives with approximately 20ms delay. The GET request in the test case 5.9 arrives with in 20ms delay on the newly established connection.

The reason for seen behavior is that the proxy tool allows web client to connect on more than one port simultaneously in an ongoing HTTP session. This result in arrival of a new request without any further delay at the web server side. Moreover, the GET request also arrives with in a delay on the connection between web server and proxy system. Therefore, it is not an appropriate result to make decision about presence of a proxy in the connection. On internet, the higher delay in initial SYN request can be a result of congestion on the network and this behavior is not persistent over the complete session. The similar behavior is seen in test case 5.10, where anonymous proxy is used to access the web page.

In test case 5.11, the Tor framework is used to access a web page hosted on the web server at lab. From figure 5.3, we see that the new connection request arrives after a delay of more than one second in initial phase of the HTTP session. The GET request on a newly established connection arrives after a delay of one second or more. The behavior of high delay in GET request remains persistent over the complete HTTP session. The figure shows IPT of incoming request packets regardless of port on which the request is made. Therefore from figure it is not clear to see higher IPT value of GET request. However, the packet trace file shows that the GET request arrives after a delay of approximately one on each newly established connection.

In Tor framework, the web client establishes a secure connection with the Tor entry node and using this single connection the web client establishes secure connections with two more middle nodes. To maintain the anonymity, the Tor framework uses only a single connection for communication using SSL protocol layer. The SSL layer provides multiplexing of the logical channels in to a single connection for more than one request from the client. The use of a single connection causes higher delay due to handling of more than one connection requests in to logical channels by multiplexing them. Moreover, the use of encryption in Tor framework results in a higher delay. The value of delay decreases, when the same Tor nodes are used again to access the web page or another web page from same server. However, the IPT value of GET packets remains around 8-10 times of RTT delay of connection between the exit Tor node and the web server. The similar behavior is noted when a proxy tool providing only a single port to the client for a HTTP session is used to

access a web page.

The conclusion from the above experiment is that a connection coming at the web server directly from a web client is distinguishable from a connection coming via Tor framework or a SOCKS proxy not providing multiple connections support to the web client. From the figure 5.3, we see that to access the same web page the number of connection request is much higher compared to other test case.

One more HTTP experiment is performed to study the data transfer mode under HTTP protocol. The result from experiment shows similar behavior as seen in the FTP-data transfer or TCP data transfer mode experiment. Therefore, if a client starts to send data towards the web server then by monitoring IPT value of incoming data packets, the presence of a proxy system in the connection can be detected.

5.2.3 The Approach Implementation for HTTP Protocol

The approach for HTTP protocol is implemented in to a script file named "tor-proxy.bro". The script can be loaded in to the Bro IDS to monitor the incoming HTTP requests at the web server. The complete source code is provided in Appendix A.5. The script checks all newly established connections through `event connection_establi shed`. If the connection is an HTTP connection then a session is created to store the session related information.

The script records IPT value of a GET request made from the web client. The values are stored in a table with the corresponding client addresses. The script monitors IPT value of the first 4 GET requests made by a web client. The average of these request values is taken and compared with the `session$rtt_cond`. If the average value is found to be higher than rtt condition then a warning is logged in to the *proxy.log*. To test performance of the script, it is run on the data files captured during the experiment. The script is loaded into Bro IDS using command as.

```
sudo bro -r http_tor_185_50ms.pcap -r http_20ms_noproxy.pcap
         -r http_anony_proxy_135_135ms.pcap -r http_tor_185_
            196ms.pcap tor-proxy
```

The result is logged in to the proxy.log file. The output contains information of any Tor connection detected in the HTTP communication. Each line in the logged entry shows time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and records IP address of the client with port number used to connect at the web server.

```
1241537538.209247: #31 Proxy Connection 195.24.77.134/40751
```

```
                                        > 129.241.209.234/http
1241538927.342802: #53 Proxy Connection 205.209.142.210/4756
                                        > 129.241.209.234/http
```

5.2.4 Application to Tor Detection

The result from HTTP protocol experiment shows that the detection of a proxy system is possible mainly when the Tor framework is used in communication. The study conducted by Mccoy et al. [38] suggested that Tor is misused for copyright infringement, hacking attempts and web page defacement. Therefore using the above approach it is possible to detect the Tor usage at the web server side by monitoring IPT value of the incoming request packets.

The current trend in malware programs is turned towards the HTTP servers for their command & control and updates. If the attacker is relaying the update data for malware through one of the HTTP server then using the given approach it is possible to detect that the incoming data is coming via a web server, which is acting as a proxy system. On detection of such proxy system in the incoming connection, an alarm can be raised and the administrator can closely monitor the connection or terminate it depending upon organization policies.

"The only thing that interferes with my learning is my education."
Albert Einstein

6 The SMTP Protocol

6.1 SMTP Protocol

The Simple Mail Transfer Protocol (SMTP) is used to transfer the email messages reliably and efficiently on the network. The SMTP uses mainly TCP as its transport protocol to deliver the messages. The SMTP can deliver a message across multiple networks using the SMTP mail relay agents. The mail relay agent temporarily stores a complete message from the another relay agent and then forwards it to the next relay agent in the direction of destination.

6.2 The Spam Messages

The spam is an unsolicited commercial email message sent by the malicious users to perform different tasks e.g. advertisement, spreading viruses etc. The spam allures malicious users due to its high speed and low cost of distribution. The spam constitutes approximate 97% [1] of total email traffic on the internet. Every day more than 30 billion messages are sent through the internet. The problem of spam is more serious as most of the spam message can actively harm the recipients by phishing frauds or malware infections.

Most of the methodologies devised to fight against the spam messages are based upon checking contents of the messages and filtering them out on the results from content inspection. The content based filtering of spam messages successfully reduce the amount of spam that actually reaches up to the user's account. One of the main tools employed this methodology to detect spam messages is SpamAssassin [13]. The Spamassasin tool incorporated several techniques to detect forged X-Mailer headers, hash busting schemes and Bayes busters [32]. However, all the mentioned methods detect spam messages based on the result from content inspection. The drawback in this approach is that the users and administrators must continually update their filtering rules. The spammers can devise new ways for altering the contents of an

email to circumvent the filters.

In addition to the content based inspection schemes, many email message filters also perform lookups in blacklists to determine whether the sender IP address is in *Blacklist* [33] or not. The blacklist is a list of known spammers, open relays and open proxies. There are more than 30 blacklists available to filter out the message based on IP address contained in blacklist. The effectiveness of blacklists is reduced to 80% due to short-lived IP addresses of the computer systems used to send spam messages.

To fight spam more effectively a new methodology is needed. The authors in [45] study the network level behavior of spams and provide information to filter them before their outspread across the internet. The intuition behind monitoring the network level behavior is that it helps to filter the spam messages near to its origin point. Whereas content based approach waits for the message to reach up to monitoring host and then runs spam test to filter it. The authors studied the network level characteristics such as: IP addresses range of hosts sending most spams, common spamming modes (e.g. open relay proxies, botnets), spamming host uptime and behavior of botnets, which are responsible for spreading spams. They suggested that most of the spams are sent by botnets and most of bots IP addresses are short-lived. The bots in botnets act as SOCKS proxies and use the system owner's credentials to forward spam messages.

6.3 SMTP Experiment

The objective for SMTP protocol experiment is to find that if a peer sending an email message is either acting as a proxy or it is a legitimate sender of the email message. The motivation for the experiment is to stop the spam messages coming from proxybots by detecting the systems acting as proxies for the spammers and dropping messages coming from them. These proxybots act as SOCKS proxies, generic TCP level proxy, for spammers and help them to hide their real IP addresses, and allow them to forward spams anonymously using legitimate IP addresses.

The experiment is performed with the three computer systems. The system C is acting as a mail client to send spam, the system P is acting as a generic TCP proxy and the system S is the mail server. The server S can be any mail relay agent used to forwards the messages to another relays or it can be the end destination for a user mail message.

In the starting, some tests are performed without any proxy system on the

network link between the server S and the client C.

$$S \xleftrightarrow{10ms} C \tag{6.1}$$

In another test, we increase the delay to study the behavioral change under different network delay conditions.

$$S \xleftrightarrow{20ms} C \tag{6.2}$$

Now we include a proxy system P in the connection between the client S and the server S under network conditions as.

$$S \xleftrightarrow{10ms} P \xleftrightarrow{40ms} C \tag{6.3}$$

Now delay between the client C and the proxy P is kept same but delay between the proxy P and the server S is increased up to 20ms.

$$S \xleftrightarrow{20ms} P \xleftrightarrow{40ms} C \tag{6.4}$$

Now delay between the client C and the proxy system P is increased to see the change in behavior at the receiving end S.

$$S \xleftrightarrow{20} P \xleftrightarrow{200ms} C \tag{6.5}$$

The Figure 6.1 illustrates result from the SMTP experiment. The figure shows IPT values of incoming SMTP request packets [34] (HELO, MAIL FROM, RCPT TO) and the ACK packets with data size equal to zero. The figure shows result from the different test cases. The result marked as "No proxy with 10ms delay" shows the test case 6.1 and the result marked with "Proxy with 10ms and 40ms delay" shows the test case 6.3. The similar notation is used for all the other test cases.

Analysis and Discussion

The result from above experiment indicates that at the receiving end by monitoring the IPT values of incoming request packets, we can distinguish whether the peer sending a message is either acting as a proxy or it is the real originator of the message. From the test 6.1, we can see that a request packet from the sender arrives with 10ms interarrival packet time or slightly higher due to the host workload delay. The similar behavior is seen in the test 6.2, where packet arrives with 20ms interarrival packet time or slightly higher. When a proxy system is included in the connection chain, we see a different behavior at the receiving end. The SMTP request packets start to arrive after higher values of IPT. In the test 6.3, we see that ACK packets from the proxy system P arrive with a IPT value of 10ms or slightly higher. Whereas the SMTP command packets arrive after 40ms or higher IPT value

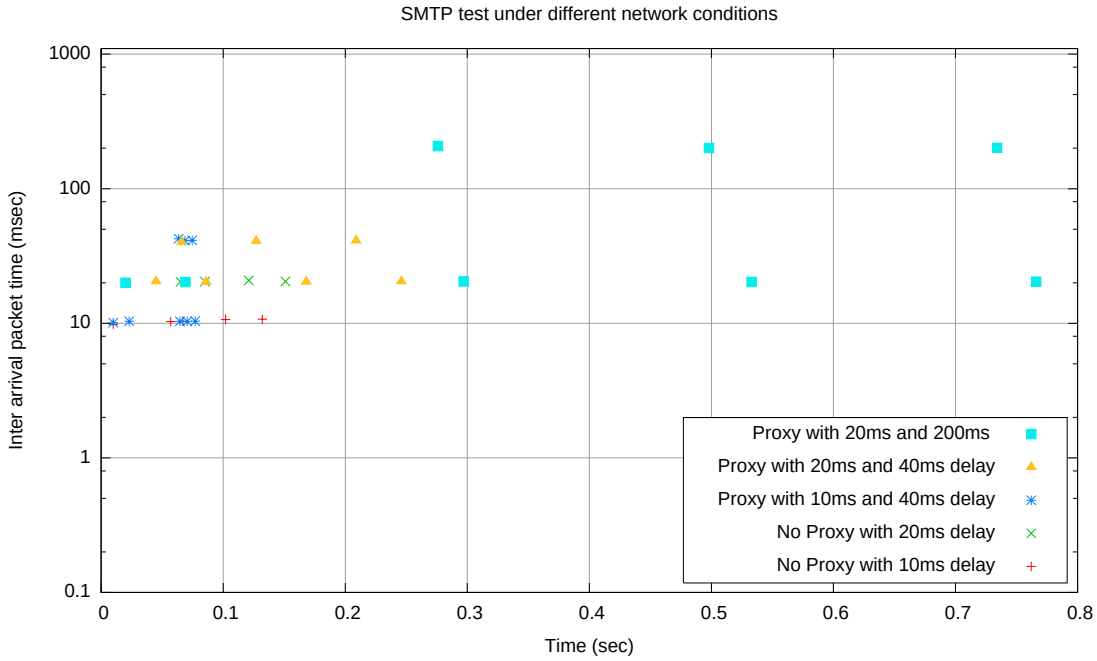

Figure 6.1: SMTP connection test result with the X-axis as time (sec) and the Y-axis as inter arrival packet time (msec) on log scale.

at the server S. The similar behavior is seen in the test 6.4 and test 6.5, where ACK packets arrive after 20ms delay and SMTP request packets arrive after 40ms and 200ms delay correspondingly.

The reason for the seen behavior is the proxy system P is acting like a generic TCP proxy and just relaying the packets in both directions. After the TCP connection is established between the server S and the proxy P, the proxy system P relays a greeting message from the server S to the client C. The greeting message travels the complete path from the server S to the client C. After relaying the greeting message, the proxy system P sends an ACK to the server S that arrives with a IPT value equals to delay between them. For example in test 6.3, the ACK packet arrives after 10ms at the server side. After receiving the greeting message, the client C sends a HELO message to the server S via the proxy system P. The HELO message travels from the client C to the server S and results in arrival at the server side after one RTT delay (10ms + 40ms) between them. Therefore, all the command packets arrive at the server side after one complete RTT delay (IPT of ACK + IPT of request) of the connection between the mail server and the mail client. The same behavior is seen in all the test cases performed under presence of a proxy system.

The conclusion from the above experiment is that a SMTP connection coming directly at the mail server can be distinguished from an incoming SMTP connection coming via proxy system. The detection is possible due to the journey of response from mail server back to the mail client and arrival of the next command at the mail server by completing the journey from the mail client to the mail server. If the proxy system first stores a message and then forwards it to the mail server then detection is not possible. Because at the mail server, the incoming SMTP connection behaves similar to a legitimate case when proxy system itself is sending a message to the mail server.

6.4 SMTP Experiment with Attachment

The SMTP experiment with an attachment is conducted to study the behavior in a case when an email message has an attachment attached to it. The experiment is performed under three different network delay conditions.

$$S \xleftrightarrow{20ms} C \tag{6.6}$$

In second test, the proxy system P is included with network conditions as

$$S \xleftrightarrow{20ms} P \xleftrightarrow{40ms} C \tag{6.7}$$

In third test, delay between the proxy P and the client C is increased up to 200ms.

$$S \xleftrightarrow{20ms} P \xleftrightarrow{200ms} C \tag{6.8}$$

The Figure 6.2 shows result from the above experiment. The result shows inter-arrival packet time of the incoming packets with data size greater than zero. The notation methodology used in this experiment is similar to the section 6.3.

Analysis and Discussion

The result from above experiment shows that the presence of a proxy system in the connection between the server S and the client C results in a distinguished behavior at the server S. In all the three test cases, we have seen that the command packets arrive according to delay between the client C and the server S. After the initial three packets (SMTP command packets) in each session, the attachment data packets start to arrive at the server side. We see that the data packets start to arrive according to RTT delay on the connection between the server S and the client C. This behavior is similar to the TCP data transfer mode experiment 3.1.2.

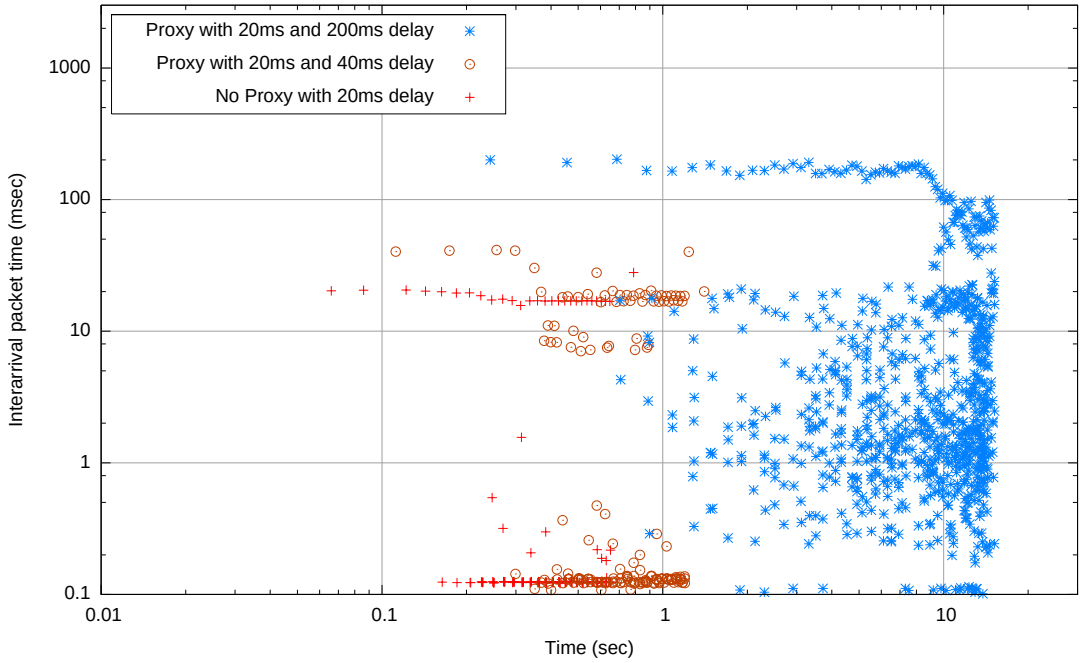

Figure 6.2: SMTP with attachment test result with the X-axis as time (sec) on log scale and the Y-axis as inter arrival packet time (msec) on log scale.

The explanation for the obtained result is the TCP window size restriction. The TCP window size limits amount of data transmission to the receiver. Initially after sending 2-3 data packets, the client C waits for an ACK to arrive before sending more data packets. This causes new data packets to arrive at the server S after a delay equal to one RTT delay on the connection between them. This behavior causes delay in new data arrival at the server end and reveals the actual delay between the server S and the client C. The similar behavior is seen in all the three test cases.

The conclusion from the SMTP attachment experiment is that at the receiving end, we can detect whether the peer sending a mail message is either acting as a proxy or is a real originator of the message. The result obtained from this experiment coupled with SMTP experiment 6.3 will result in detection of spam messages with high confidence, when spam messages are sent by proxy bots.

6.5 The Approach Implementation for SMTP Protocol

The proposed approach to detect the spam messages is implemented in a script file named "smtp-proxy.bro". The script can be loaded in the Bro IDS to monitor IPT values of the incoming SMTP connections. The complete source code is given in Appendix A.6. The script monitors all newly established TCP connection and if the connection is a SMTP connection then the scripts start to monitor IPT values of the incoming request packets.

The script uses a SMTP session record specified in a "smtp.bro" script provided with the Bro IDS. Some additions are done to the "smtp.bro" to store the IPT values and related information. A patch is provided with the report for all the modifications. The script records IPT values of incoming SMTP request packets, which have value higher than the RTT conditions. After the first four request packets are received at the mail server, the script takes average of the IPT values and compares it with the RTT condition. If the average value is found higher than the RTT condition then a warning is logged in to the *proxy.log*. The script is run on the data files captured during the experiment to detect spam messages coming from the proxy system. The script is loaded in to the Bro IDS and following data files are used to test the script.

```
sudo bro -r smtp_20_200_proxy_att.pcap -r smtp_20_noproxy.pcap
         -r smtp_20_40_proxy.pcap mt -r smtp_20_200_proxy.pcap
            smtp-proxy
```

The result obtained from running the script on above data files is logged in to the proxy.log file. Each line in the logged entry shows time on which a connection is marked as a proxy connection. It shows the session ID for that specific session and records IP address of the client with port number used to connect at the mail server.

```
1241088659.242469 #1 Proxy Connection: 129.241.209.164/58208
                                         > 129.241.208.198/smtp
1241107929.170802 #3 Proxy Connection: 129.241.209.164/42145
                                         > 129.241.208.198/smtp
1244108153.509671 #4 Proxy Connection: 129.241.209.234/45166
                                         > 129.241.208.186/smtp
```

6.6 Application to Spam Detection

Currently in the internet, there are thousands of active proxybots which are helping the spammer to send spam messages in to the network. The result from SMTP

protocol experiments shows that at the mail server, it is possible to detect the presence of proxybots in an incoming request. Therefore, the approach can be applied for detection of the spam messages coming from such proxybots. If the test about proxybots shows positive result then the incoming message is most likely a spam and can be dropped. The given approach mainly works near to the spam originating point and helps to detect spam before its outspread in the network.

The study conducted by Ramachandran et. al [45] suggests that most of bots in the botnets have short lived IP addresses and this causes a decrease in efficiency of blacklist used to detect the open proxy systems. Therefore, the above approach can detect such proxybots in the botnets without any need for previous information. If a mail server receives many spam messages from a computer system and behavior is persistent, then its IP address can be added in to the blacklists to avoid spam messages from it in the future. The given approach can be coupled with the existing approaches to detect spams e.g. Spamassassin [13] and lead to almost zero percent false positive rate in detection of the spam messages.

"If you do not change direction, you may end up where you are heading."
Lao Tzu

7 Conclusion and Future Work

7.1 General Discussion

The result from experiments performed on different application layer protocols showed that on the receiver side, it is possible to distinguish from an incoming TCP connection whether the peer is acting as a proxy system or the peer is a real originator of the connection. The distinction depends upon two main factors.

- **Application Protocol Type :** The application layer protocols have its own characteristics. These characteristics result in different behavior at the receiver end under different network conditions. For example, the SSH protocol behaves differently than Telnet due to its custom window size. However, during bulk data transfer mode similar behavior was noticed, irrespective of any application protocol used to transfer data.
- **Proxy Type :** the type of proxy used in the communication session results in distinct behavior at the receiver end. From the experiment conducted on different application protocols, we saw that when a generic TCP proxy is used on the proxy system, then the actual session is setup between the server and the client and not between the server and the proxy system. This results in detection of a proxy system in the early phase of session setup as a message will travel from the server to the client. If the application protocol itself is used at a proxy system then it results in a session establishment between the server and the proxy system and behavior of IPT values at the server side is similar to a legitimate case.

From the study conducted in this master dissertation, a conclusion was drawn when more than one proxy systems are present in the network path. The behavior in the given case will be different under the different application layer protocols and the type of the proxy used. If the generic TCP proxy is used in remote login protocols then on the receiving end, delay will be the sum of delays of all connections in the connection chain up to the receiver. Whereas if an application level proxy is

used in remote login protocols, then observed delay at the receiver end will be equal to the maximum delay of any connection in the connection chain.

In the FTP protocol, we have seen that delay noticed at the receiver side is equal to the sum of delays of all connections in the connection chain. The SMTP protocol showed similar behavior if more than one proxy system is present in the connection chain. In the bulk data transfer mode, delay observed at the receiver side will be equal to the maximum delay of any connection in the connection chain.

7.2 Conclusion

The goal of this master dissertation was to find if it possible for a receiver to decide whether the other end of a TCP connection is a real originator of the connection or merely acting as an intermediary host in the connection chain. The attackers relay their attack through intermediary hosts to hide their true IP addresses and remain anonymous on the network. The use of intermediary hosts helps them from being traced and gets punished.

This master dissertation proposes an approach, which enables a receiver to detect whether the peer communicating on a TCP connection is a real originator of the connection or merely a proxy system in the connection chain. The approach relies upon the use of Nagle algorithm and the TCP congestion control algorithms to transfer data on a TCP connection. The approach monitors the *Interarrival packet time* value of incoming packets at the receiver and based upon this information a decision is made. The approach was tested on different application layer protocols and found effective on most of them. The approach can be used in many applications like Manual intrusion detection to detect the break in attempts, Tor usage detection to avoid the copyright infringement or web page defacement and detection of a spam message when the message is received from proxy bots.

The approach can be coupled with the existing methods to detect network based attacks. This coupling with existing methods will result in a very high confidence in detection of such network based attacks and decrease in false warning rate. For example, if a spam filter detects an incoming message as a spam but with some uncertainty then using the proposed approach, the source of incoming message can be checked. If the source appears to be a proxy system then incoming message is a spam without any uncertainty.

The approach is able to detect the proxy system on an incoming connection, as long as the attacker uses intermediary hosts to relay his traffic. However, if the

attacker uses intermediary host in store-and-forward way during the attack then detection is not possible. The later case will result in a similar behavior as the last intermediary host initiates a communication in a legitimate case.

7.3 Future Work

In this study, the experiments were performed on a test bed in the laboratory. It would be an interesting work to study performance of the proposed approach in a large scale network such as an ISP network.

The implementation provided for the proposed approach is in a beta version. In the current implementation, the Round Trip Time (RTT) of an incoming connection is calculated during the initial TCP 3-way handshake. However, the RTT value can vary in the network during the lifetime of a session e.g. due to occurrence of a congestion in the network. It would be a good approach to calculate RTT value, which reflects current network conditions. The current value of RTT will also result in less false positive rate.

It would also be an interesting idea to study a change in latency behavior at the receiving end, when either a sender or a proxy system is loaded. The load can be caused either by a sudden increase of work load or by a denial of service attack on the system.

The approach can be applied to more applications, which use proxy systems. It would be an interesting work to find more applications, and test the given approach on them. One such application is the digital forensics investigation. During the investigation of an attack, the peer from which TCP connection is received; becomes a victim of suspicion. The investigator can use the proposed approach and study available traffic traces to decide whether the peer actually launched an attack or was merely used as a stepping stone by the attacker.

The approach is going to be used in the OISF's next generation Intrusion Detection System [2].

A

Appendix A

A.1 Manual Intrusion Detection in SSH Protocol

```
1  # The given code can detect the presence of a proxy system,
2  # in the inital key exchange phase. This will detect the
3  # presence of only generic TCP proxy.
4
5  @load mt
6
7  type ssh_session_info: record {
8      id: count;
9      connection_id: conn_id;
10     rtt: interval;
11     rtt_cond: interval;
12     last_pkt_t: time;
13     pkt_t: time;
14     pkt_cnt: count;
15     count_p: count;
16     sum: interval;
17     avg_t: interval;
18     alarmp: bool;
19     };
20
21 global ssh_sessions: table[conn_id] of ssh_session_info;
22 global ssh_session_id = 0;
23 const proxy_file = open_log_file("proxy") &redef;
24
25 function new_ssh_session(c: connection)
26     {
27     local session = c$id;
28     local new_id = ++ssh_session_id;
```

```
29      local info: ssh_session_info;
30      info$id = new_id;
31      info$connection_id = session;
32      info$rtt = 0sec;
33      info$rtt_cond = 0sec;
34      info$last_pkt_t = c$start_time;
35      info$pkt_t = c$start_time;
36      info$pkt_cnt = 0;
37      info$count_p = 0;
38      info$sum = 0 sec;
39      info$avg_t = 0 sec;
40      info$alarmp = F;
41
42      ssh_sessions[session] = info;
43      }
44
45  function is_ssh_conn(c: connection): bool
46      {
47      return c$id$resp_p == 22/tcp ;
48      }
49
50  event new_packet (c: connection, p: pkt_hdr)
51      {
52      local id = c$id;
53      if ( id !in ssh_sessions )
54          return;
55      local session = ssh_sessions[id];
56      session$rtt = connection_rtt(c$id);
57      if (session$rtt > .05sec)
58          session$rtt_cond =  session$rtt*1.25;
59      else
60          session$rtt_cond =  session$rtt*2.5;
61      if (p$ip$src == id$orig_h && p$tcp$flags != TH_ACK && ↩
           session$pkt_cnt < 7)
62          {
63          session$pkt_t = network_time();
64          ++session$pkt_cnt;
65          if((session$pkt_t - session$last_pkt_t) > ↩
              session$rtt_cond )
66              {
67              session$sum = session$sum + (session$pkt_t - ↩
                  session$last_pkt_t);
68              ++session$count_p;
```

```
69              if(session$pkt_cnt >= 7 && ! session$alarmp)
70                  {
71                  session$avg_t = (session$sum/(session$count_p))↩
                        ;
72                  if(session$avg_t > session$rtt_cond)
73                      {
74                      print proxy_file, fmt ("%.6f: #%s Proxy ↩
                            Connection %s", network_time(), ↩
                            prefixed_id(session$id), id_string(c$id)↩
                            );
75                      session$alarmp = T;
76                      }
77                  }
78              }
79          }
80      session$last_pkt_t = network_time();
81      }
82
83  event connection_established(c: connection)
84      {
85      if ( is_ssh_conn(c) )
86          {
87          local id = c$id;
88          if ( id !in ssh_sessions )
89              new_ssh_session(c);
90          }
91      }
```

A.2 Manual Intrusion Detection in Telnet Protocol

```
1  @load mt
2
3
4  type telnet_session_info: record {
5          id: count;
6          state: bool;
7          connection_id: conn_id;
8          rtt: interval;
9          rtt_cond: interval;
10         first_pkt_t: time;
11         last_pkt_t: time;
```

```
12          pkt_t: time;
13
14          chkcond_f: bool;
15          count_p: count;
16          sum: interval;
17          avg_t: interval;
18
19          alarmp: bool;
20          pkt_cnt: count;
21          nwpchk_f: bool;
22          proxycond_f: bool;
23      };
24
25
26  global telnet_sessions: table[conn_id] of telnet_session_info;
27  const rtt_threshold = 1.2 sec;
28  global first_pkt_t: time;
29  global type_speed = .25sec;
30  global type_brk = .7sec;
31  global telnet_session_id = 0;
32  global arrival_t : table[conn_id, count] of interval;
33  const proxy_file = open_log_file("proxy") &redef;
34
35
36  function new_telnet_session(c: connection)
37      {
38      local session = c$id;
39      local new_id = ++telnet_session_id;
40
41      local info: telnet_session_info;
42      info$id = new_id;
43      info$state = F;
44      info$connection_id = session;
45      info$rtt = 0sec;
46      info$rtt_cond = 0sec;
47      info$first_pkt_t = c$start_time;
48      info$last_pkt_t = c$start_time;
49      info$pkt_t = c$start_time;
50
51      info$alarmp = F;
52      info$chkcond_f = F;
53      info$count_p = 0;
54      info$sum = 0 sec;
```

```
55      info$avg_t = 0 sec;
56      info$pkt_cnt = 1;
57      info$nwpchk_f = F;
58      info$proxycond_f = T;
59
60      telnet_sessions[session] = info;
61      }
62
63  function is_telnet_conn(c: connection): bool
64      {
65      return c$id$resp_p == telnet || c$id$resp_p == rlogin;
66      }
67
68  event login_success(c: connection, user: string, client_user: ↩
        string,
69              password: string, line: string)
70      {
71      local id = c$id;
72      local session = telnet_sessions[id];
73      session$state= T;
74      }
75
76  event new_packet (c: connection, p: pkt_hdr)
77      {
78
79      local id = c$id;
80      if ( id !in telnet_sessions )
81          return;
82
83      local session = telnet_sessions[id];
84
85      if(!session$state)
86      {
87      session$rtt = connection_rtt(c$id);
88      if (session$rtt > .05sec)
89          session$rtt_cond = (session$rtt*1.25);
90      else
91          session$rtt_cond = (session$rtt*2);
92      type_brk = type_brk + session$rtt_cond;
93
94      if (p$ip$src == id$orig_h && p$tcp$flags != TH_ACK)
95          {
96          if(session$count_p < 4 && !session$nwpchk_f)
```

```
97              {
98              session$pkt_t = network_time();
99              session$sum = session$sum + (session$pkt_t - ↩
                   session$last_pkt_t);
100             ++session$count_p;
101             ++session$pkt_cnt;
102         }
103         if(session$count_p >= 4 && ! session$nwpchk_f)
104             {
105             session$avg_t = (session$sum/(session$count_p));
106             if(session$avg_t > session$rtt_cond)
107                 {
108                 print proxy_file, fmt("%.6f: #%s Proxy ↩
                       Connection using generic TCP proxy %s", ↩
                       network_time(), prefixed_id(session$id), ↩
                       id_string(c$id));
109                 session$alarmp = T;
110                 }
111             session$nwpchk_f = T;
112             }
113         if(session$nwpchk_f)
114             {
115             session$sum = 0 sec;
116             session$count_p = 0;
117             session$pkt_cnt = 0;
118             session$avg_t = 0 sec;
119             session$pkt_t = c$start_time;
120             session$last_pkt_t = c$start_time;
121             }
122         }
123         session$last_pkt_t = network_time();
124         return;
125     }
126
127     if (p$ip$src == id$orig_h && p$tcp$flags != TH_ACK)
128         {
129             session$pkt_t = network_time();
130             if( ((session$pkt_t - session$first_pkt_t) < 90sec ↩
                   && session$pkt_cnt < 10) && ! (p$tcp$flags == ↩
                   TH_ACK + TH_FIN) )
131             {
132                 if((session$pkt_t - session$last_pkt_t) > ↩
                       rtt_threshold)
```

```
133                     session$last_pkt_t = network_time();
134
135                 if((session$pkt_t - session$last_pkt_t) > 0sec ↩
                       && (session$pkt_t - session$last_pkt_t) < ↩
                       type_brk)
136                 {
137                     session$sum = session$sum + (session$pkt_t ↩
                           - session$last_pkt_t);
138                     ++session$count_p;
139                     ++session$pkt_cnt;
140                 }
141                 session$chkcond_f = T;
142             }
143
144             if ( (!session$chkcond_f || p$tcp$flags == TH_ACK +↩
                    TH_FIN) && session$proxycond_f)
145             {
146             session$avg_t = (session$sum/(session$count_p - 1))↩
                   ;
147             session$rtt_cond = session$rtt_cond + type_speed;
148             if(session$avg_t > session$rtt_cond && ! ↩
                   session$alarmp)
149                 {
150                     print proxy_file, fmt ("%.6f: #%s Proxy ↩
                           Connection %s", network_time(), ↩
                           prefixed_id(session$id), id_string(c$id)↩
                           );
151                 }
152             session$proxycond_f = F;
153             }
154             session$last_pkt_t = network_time();
155             session$chkcond_f = F;
156         }
157     }
158
159 event connection_established(c: connection)
160     {
161     if ( is_telnet_conn(c) )
162         {
163         local id = c$id;
164         if ( id !in telnet_sessions )
165             new_telnet_session(c);
166         }
```

```
167     }
```

A.3 Manual Intrusion Detection in FTP-Control Protocol

```
1  # The given code can detect the presence of intermediary hosts
2  # in the incoming FTP commands by monitoring the IPT values
3  # of requests.
4
5  @load ftp
6
7      const proxy_file = open_log_file("proxy") &redef;
8      global rtt: interval;
9      global rtt_cond: interval;
10     const rtt_threshold = 1.2 sec;
11 # Event generated by Bro when a ftp request is received from ↩
       ftp client
12 event ftp_request(c: connection, command: string, arg: string)
13     {
14     local request = arg == "" ? command : cat(command, " ", arg↩
           );
15     rtt = connection_rtt(c$id);
16         if (rtt > .05sec)
17             rtt_cond = rtt*1.25;
18         else
19             rtt_cond = rtt*2;
20     local id = c$id;
21     local session = FTP::ftp_sessions[id];
22     session$request = request;
23     ++session$num_req;
24     session$request_t = network_time();
25     if ((session$request_t - session$last_request_t) > rtt_cond↩
           && (session$request_t - session$last_request_t) < ↩
          rtt_threshold)
26         session$sum = session$sum + (session$request_t - ↩
              session$last_request_t);
27     if ((session$num_req > 4) && (session$sum/session$num_req) ↩
          > rtt_cond && !session$alarmp)
28         {
29         print proxy_file, fmt("%.6f #%s Proxy Connection: %s",
30             network_time(), prefixed_id(session$id), id_string(↩
                c$id));
```

```
31          session$alarmp = T;
32          }
33      session$last_request_t = network_time();
34      }
```

A.4 Intermediary Hosts Detection in FTP-Data Protocol

```
1  # The code below can detect the presence of intermediary hosts
2  # in the incoming data from a peer. The code monitors mainly
3  # the IPT values which are higher than the RTT condition.
4
5  @load mt
6
7  type ftpdata_session_info: record {
8      id: count;
9      connection_id: conn_id;
10     rtt: interval;
11     rtt_cond: interval;
12     last_pkt_t: time;
13     pkt_t: time;
14     count_p: count;
15     sum: interval;
16     avg_t: interval;
17     alarmp: bool;
18     };
19
20 global ftpdata_sessions: table[conn_id] of ftpdata_session_info↩
       ;
21 global ftpdata_session_id = 0;
22 const proxy_file = open_log_file("proxy") &redef;
23
24 function new_ftpdata_session(c: connection)
25     {
26     local session = c$id;
27     local new_id = ++ftpdata_session_id;
28     local info: ftpdata_session_info;
29     info$id = new_id;
30     info$connection_id = session;
31     info$rtt = 0sec;
32     info$rtt_cond = 0sec;
33     info$last_pkt_t = c$start_time;
```

```
34      info$pkt_t = c$start_time;
35      info$count_p = 0;
36      info$sum = 0 sec;
37      info$avg_t = 0 sec;
38      info$alarmp = F;
39
40      ftpdata_sessions[session] = info;
41      }
42
43  function is_ftpdata_conn(c: connection): bool
44      {
45      return c$id$orig_p == 20/tcp ;
46      }
47
48  event new_packet (c: connection, p: pkt_hdr)
49      {
50
51      local id = c$id;
52      if ( id !in ftpdata_sessions )
53          return;
54      local session = ftpdata_sessions[id];
55      if (p$ip$src == id$resp_h && p$tcp$flags == 24)
56          {
57          session$pkt_t = network_time();
58          session$rtt = connection_rtt(c$id);
59          if (session$rtt > .05sec)
60              session$rtt_cond =  session$rtt*1.25;
61          else
62              session$rtt_cond =  session$rtt*2.5;
63
64          if((session$pkt_t - session$last_pkt_t) > ↩
               session$rtt_cond )
65              {
66              session$sum = session$sum + (session$pkt_t - ↩
                   session$last_pkt_t);
67              ++session$count_p;
68              if(session$count_p > 5 && ! session$alarmp)
69                  {
70                  session$avg_t = (session$sum/(session$count_p))↩
                       ;
71                  if(session$avg_t > session$rtt_cond)
72                      {
73                      print proxy_file, fmt ("%.6f: #%s Proxy ↩
```

```
                        Connection %s", network_time(), ↩
                        prefixed_id(session$id), id_string(c$id)↩
                        );
74                    session$alarmp = T;
75                    }
76                }
77            }
78        }
79    session$last_pkt_t = network_time();
80    }
81
82 event connection_established(c: connection)
83    {
84    if ( is_ftpdata_conn(c) )
85        {
86        local id = c$id;
87        if ( id !in ftpdata_sessions )
88            new_ftpdata_session(c);
89        }
90    }
```

A.5 Tor Detection in HTTP Protocol

```
1  # The given code can detect the Tor usage in an incoming HTTP
2  # request at the web server. The IPT value of first four GET
3  # requests are analyzed to check the presence of proxy system.
4
5  @load mt
6
7  # HTTP Session record to store the IPT values, Id, seesion info↩
       .
8  type http_session_info: record {
9      id: count;
10     connection_id: conn_id;
11     rtt: interval;
12     rtt_cond: interval;
13     last_pkt_t: time;
14     pkt_t: time;
15     };
16 # HTTP GET request record to store the GET request IPT value, ↩
       client add, no. of GET request.
17 type get_request: record {
```

```
18      get_time: interval;
19      client_addr: addr;
20      get_cnt: count;
21      get_avg: interval;
22      alarmp: bool;
23      };
24
25  global http_get_requests: table[addr] of get_request;
26  global http_sessions: table[conn_id] of http_session_info;
27  global http_session_id = 0;
28  const proxy_file = open_log_file("proxy") &redef;
29  # Function to initialize the new HTTP session
30  function new_http_session(c: connection)
31      {
32      local session = c$id;
33      local new_id = ++http_session_id;
34      local info: http_session_info;
35      info$id = new_id;
36      info$connection_id = session;
37      info$rtt = 0sec;
38      info$rtt_cond = 0sec;
39      info$last_pkt_t = c$start_time;
40      info$pkt_t = c$start_time;
41
42      http_sessions[session] = info;
43      }
44
45  function is_http_conn(c: connection): bool
46      {
47      return c$id$resp_p == 80/tcp ;
48      }
49  # Function to initialize the new GET request
50  function new_http_get(c: connection)
51      {
52      local client_a = c$id$orig_h;
53      local req: get_request;
54
55      req$get_time = 0 sec;
56      req$client_addr = client_a;
57      req$get_cnt = 1;
58      req$get_avg = 0 sec;
59      req$alarmp = F;
60
```

```
61      http_get_requests[client_a] = req;
62      }
63
64  event new_packet (c: connection, p: pkt_hdr)
65      {
66
67      local id = c$id;
68      if ( id !in http_sessions )
69          return;
70      local session = http_sessions[id];
71      if(p$ip$src == id$orig_h && p$tcp$flags == 16)
72          session$last_pkt_t = network_time();
73      if (p$ip$src == id$orig_h && p$tcp$flags == 24)
74          {
75          session$pkt_t = network_time();
76          if (p$ip$src !in http_get_requests)
77              new_http_get(c);
78          local get_requestl = http_get_requests[p$ip$src];
79          get_requestl$get_time = get_requestl$get_time + (↩
               session$pkt_t - session$last_pkt_t);
80          ++get_requestl$get_cnt;
81          if(get_requestl$get_cnt > 4 && !get_requestl$alarmp)
82              {
83              get_requestl$get_avg = get_requestl$get_time/↩
                   get_requestl$get_cnt;
84              session$rtt = connection_rtt(c$id);
85              if (session$rtt > .05sec)
86                  session$rtt_cond =  session$rtt*1.25;
87              else
88                  session$rtt_cond =  session$rtt*2.5;
89              if (get_requestl$get_avg > session$rtt_cond)
90                  print proxy_file, fmt ("%.6f: #%s Proxy ↩
                       Connection %s", network_time(), prefixed_id(↩
                       session$id), id_string(c$id));
91              get_requestl$alarmp = T;
92              }
93
94          }
95      }
96
97  event connection_established(c: connection)
98      {
99      if ( is_http_conn(c) )
```

```
100         {
101         local id = c$id;
102         if ( id !in http_sessions )
103             new_http_session(c);
104         }
105     }
```

A.6 Spam Detection in SMTP Protocol

```
1  # The code below can detect the presence of a proxybots
2  # by monitoring the IPT values of incoming SMTP requests.
3
4  @load smtp
5
6      const proxy_file = open_log_file("proxy") &redef;
7      global rtt: interval;
8      global rtt_cond: interval;
9  # Event generated by Bro IDS when a SMTP request is received ↩
       from client
10 event smtp_request(c: connection, is_orig: bool, command: ↩
       string, arg: string)
11     {
12     local request = arg == "" ? command : cat(command, " ", arg↩
           );
13     rtt = connection_rtt(c$id);
14     if (rtt > .05sec)
15         rtt_cond = rtt*1.25;
16     else
17         rtt_cond = rtt*2;
18     local id = c$id;
19     local session = SMTP::smtp_sessions[id];
20     if(session$origin_h == id$orig_h)
21         ++session$num_req;
22     session$request_t = network_time();
23     if ((session$request_t - session$last_request_t) > rtt_cond↩
           )
24         session$sum = session$sum + (session$request_t - ↩
               session$last_request_t);
25     if ((session$num_req > 4) && (session$sum/(session$num_req)↩
           ) > rtt_cond && !session$alarmp)
26         {
27         print proxy_file, fmt("%.6f #%s Proxy Connection: %s ",↩
```

```
            network_time(), prefixed_id(session$id), id_string(↩
            c$id));
28          session$alarmp = T;
29          }
30      session$last_request_t = network_time();
31      }
```

B
Appendix B

B.1 Upload Folder Contents

The Upload folder contains three subfolders: -

1. **Experiment_Data:** This folder contains captured traffic files from all the experiments performed in this thesis work. The files are classified according to the chapter numbers. In each chapter folder, there are subfolders corresponding to the experiment of different application protocols.

2. **Implementation_code:** This folder contains the implementation code for all the application protocols. The folder also contains the patch files for "conn.bro, ftp.bro and smtp.bro" files.

3. **Scripts_used_in_experiments:** This folder contains the script used to perform the experiments for different protocols. Most of the scripts used to perform experiments are written by the author himself. They are classified according to the chapter numbers. The Chapter_3 folder contains TG script used to generate TCP traffic and a script to act as a server to receive TCP traffic. The Chapter_4 folder contains Expect script used to perform SSH and Telnet experiments. The Chapter_5 folder contains a script used as a web proxy in HTTP experiment.

References

[1] Security intelligence technical report. *Microsoft*, July-Dec, 2008. 1, 65

[2] Open information security foundation. *http://www.openinfosecfoundation.org/*, Last Accessed June, 2009. 75

[3] Bro reference manual. *http://www.bro-ids.org/wiki/index.php/Reference_Manual*, Last Accessed May, 2009. 40

[4] Daemonlogger, packet logger & soft tap. *http://www.snort.org/users/roesch/Site/Daemonlogger/Daemonlogger.html*, Last Accessed May, 2009. 15, 19

[5] Detecting tor users. *http://ha.ckers.org/weird/privoxy-test.html*, Last Accessed May, 2009. 3

[6] Deter - testbed for cyber security research. *http://www.isi.deterlab.net/*, Last Accessed May, 2009. 11

[7] The gnu netcat project. *http://netcat.sourceforge.net/*, Last Accessed May, 2009. 13, 19

[8] High performance packet capture. *http://www.net.t-labs.tu-berlin.de/research/hppc/*, Last Accessed May, 2009. 15

[9] Libpcap - packet capture library. *http://www.tcpdump.org/*, Last Accessed May, 2009. 15, 17

[10] Metasploit decloaking engine. *http://decloak.net/*, Last Accessed May, 2009. 3

[11] Proxy server - forensic wiki. *http://www.forensicswiki.org/wiki/Proxy_server*, Last Accessed May, 2009. 3

[12] Report of nsf workshop on network research testbeds. *http://www-net.cs.umass.edu/testbed_workshop/*, Last Accessed May, 2009. 11

[13] Spam assassin. *http://spamassassin.apache.org*, Last Accessed May, 2009. 3, 65, 72

[14] Traffic generator tool. *http://www.postel.org/tg/*, Last Accessed May, 2009. 12, 19

[15] Wireshark packet analyzer. *http://www.wireshark.org/docs/wsug_html_chunked/index.html*, Last Accessed May 2009. 15

[16] J. Touch (1995). Dartnet overview. *http://www.isi.edu/touch/dli95/dartnet-dli.html*, Last Accessed May, 2009. 11

[17] M. Allman, S. Floyd, and C. Partridge. Increasing tcp's initial window, 2002. 8

[18] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, 1999. 8

[19] Martin Arlitt, Balachander Krishnamurthy, and Jeffrey C. Mogul. Predicting short-transfer latency from tcp arcana: a trace-based validation. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association. 32

[20] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association. 11

[21] Avrim Blum, Dawn Song, and Shobha Venkataraman. Detection of interactive stepping stones: Algorithms and confidence bounds. In *in Conference of Recent Advance in Intrusion Detection (RAID), (Sophia Antipolis, French Riviera*, pages 258–277. Springer, 2004. 3, 11

[22] Mark Carson and Darrin Santay. Nist net: A linux-based network emulation tool. *Computer Communication Review*, 33:2003, 2003. 13

[23] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, 1981. 59

[24] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 2–15, 2003. 59

[25] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. 59

[26] Marc Donner. Cyberassault on estonia. *IEEE security & privacy*, 2007. 1

[27] David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson Jason Coit, , Stuart Staniford, Jason Coit, and Stuart Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting

maximum tolerable delay. In *in Proc. of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 17–35. Springer, 2002. 3, 10

[28] D.Moore et al. *The Spread of the Sapphire/Slammer Worm.* CAIDA Technical Report, `http://www.caida.org/publications/papers/2003/sapphire/sapphire.html`, 2003. 1

[29] Jørg Wallerich Fabian Schneider and Anja Feldmann. Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. *Springer-Verlag Berlin Heidelberg, LNCS 4427*, 2007. 15

[30] Ceki Gulcu and Gene Tsudik. Mixing email with babel. In *Symposium on Network and Distributed System Security*, pages 2–16, 1996. 59

[31] Stephen Hemminger. Network emulation with netem. *Open Source Development Lab*, 2005. 13, 19

[32] J.Mason. Spam forensics: Reverse-engineering spammer tactics. *http://spamassassin.apache.org/presentations/2004-09-Toorcon/html/*, Last Accessed May, 2009. 65

[33] William K. Blacklists, blocklists, dnsbl's, and survival. *http://www.scconsult.com/bill/dnsblhelp.html*, Last Accessed May, 2009. 3, 66

[34] J. Klensin. Simple mail transfer protocol, rfc 5321, October, 2008. 67

[35] Felix Leder and Tillmann Werner. Know your enemy: Containing conficker "to tame a malware". *The Honeynet Project*, April, 2009. 1

[36] Don Libes. Expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Summer 1990 USENIX Conference*, pages 183–192, 1990. 16, 34

[37] Steven McCanne and Van Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association. 17

[38] Damon Mccoy, Kevin Bauer, Dirk Grunwald, Kohno Tadayoshi, and Douglas Sicker. Shining light in dark places: Understanding the tor network. In *PETS '08: Proceedings of the 8th international symposium on Privacy Enhancing Technologies*, pages 63–76, Berlin, Heidelberg, 2008. Springer-Verlag. 1, 59, 64

[39] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, jan 1984. 7

[40] Vern Paxson. Empirically derived analytic models of wide-area tcp connections. *IEEE/ACM Trans. Netw.*, 2(4):316–336, 1994. 32

[41] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1998. 17, 40

[42] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3:226–244, 1995. 32

[43] Pai Peng Peng, Peng Ning, and Douglas S. Reeves. Active timing-based correlation of perturbed traffic flows with chaff packets. In *In Proceedings of 2nd International Workshop on Security in Distributed Computing Systems (SDCS*, pages 107–113, 2005. 11

[44] J. Postel and J. Reynolds. File transfer protocol (ftp), rfc 959, October 1985. 53

[45] Anirudh Ramachandran and Nick Feamster. Understanding the network-level behavior of spammers. *SIGCOMM Comput. Commun. Rev.*, 36(4):291–302, 2006. 2, 3, 66, 72

[46] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16:482–494, 1998. 59

[47] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27:31–41, 1997. 13

[48] Nick Mathewson Roger Dingledine and Paul Syverson. Tor: The second-generation onion router. In *In Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004. 1, 59

[49] Lee M. Rossey, Robert K. Cunningham, David J. Fried, Jesse C. Rabek, Richard P. Lippmann, Joshua W. Haines, and Marc A. Zissman. Lariat: Lincoln adaptable real-time information assurance testbed. In *In IEEE Proc. Aerospace Conference*, pages 2671–2682, 2001. 12

[50] Stuart Staniford-Chen and L. Todd Heberlein. Holding intruders accountable on the internet, 1994. 3, 9

[51] J. Turner. First gigabit kits workshop. *http://www.arl.wustl.edu/projects/archive/gigabitkits/Workshop_99_07/slides/turnerintro.pdf*, Last Accessed May, 2009. 11

[52] Xinyuan Wang and Douglas S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 20–29, New York, NY, USA, 2003. ACM. 3, 10

[53] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *In Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, 2002. 11

[54] T. Ylonen and C. Lonvick. The secure shell (ssh) connection protocol, rfc 4254, 2006. 37

[55] T. Ylonen and C. Lonvick. The secure shell (ssh) protocol architecture, rfc 4251, 2006. 33

[56] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol, rfc 4253, 2006. 35

[57] Kunikazu Yoda and Hiroaki Etoh. Finding a connection chain for tracing intruders. In *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*, pages 191–205, London, UK, 2000. Springer-Verlag. 3, 10

[58] Yin Zhang and Vern Paxson. Detecting stepping stones. In *In Proceedings of the 9th USENIX Security Symposium*, pages 171–184, 2000. 3, 10

[59] Johnson Alan Zhang Linfeng, Persaud Anthony and Guan Yong. Stepping stone attack attribution in non-cooperative ip networks. In *IOWA State University Tech. Report TR-2005-02-1*, Feb, 2005. 11