NTNU
Norwegian University of
Science and Technology

Experimental evaluation of DDoS detection and prevention using open-source and commodity hardware

Meklit Elfiyos Dekita

Master of Science in Telematics - Communication Networks and Networked
Submission date: March 2018
Supervisor: Yuming Jiang, IIK
Co-supervisor: Ivar Arnesen, Ivar Arnesen invest AS

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

.

Title: Experimental Evaluation of Inline DDoS Detection and Prevention using Open-source Solutions and Commodity Hardware

Student: Meklit Elfiyos Dekita

Problem description:

Distributed Denial of Service (DDoS) is one of the rapidly growing attacks posing a significant threat to internet resources. If a DDoS attack is not handled during the initial states, the attack may result in service unavailability and has potential costly consequences. There are different DDoS detection and prevention mechanisms. If we could compare a live network traffic with a pattern of normal network traffic, alerts could be raised and filters applied to filter away any potential packet storm. Considering the ongoing improvements in modern commodity hardware and software architecture, there is a tremendous power for further network traffic processing. Combining this with online available, flexible and cost-effective open source DDoS detection tools can be an effective solution. Putting that into consideration, this thesis will mainly study the possibilities and performance of DDoS detection and prevention on commodity server using open source solutions. An experimental testbed will be setup and evaluation of the proposed solution will be conducted using that testbed.

Responsible professor: Yuming Jiang, IIK

Supervisor: Ivar Arnesen, Ivar Arnesen Invest AS

Abstract

Distributed Denial of Service (DDoS) attack is a serious threat to companies with an active online business as its scope is increasing in size, frequency and complexity. That is why it has become a high priority task to prevent DDoS attack for the internet stakeholders. The complexity of DDoS attacks makes their detection and mitigation difficult. Moreover, the high operational costs to deploy mitigation solutions makes deployment at the edge of victim networks not cost-effective. On the other hand, improvements in modern commodity hardware and software architecture exhibit tremendous power to process network traffics. Combining this with online available, flexible and effective open source DDoS detection tools can give an efficient solution to mitigate DDoS attacks.

The goal of this research is to study the possibilities and performance of DDoS detection and prevention on commodity hardware using open source solutions. The experiment is carried out in the implemented experimental DDoS detection testbed. Based on findings during the work of this thesis, we have come to the conclusion that using commodity hardware with effective DDoS detection application like fastnetmon and improved fast packet capturing frameworks such as netmap and PF_Ring ZC, has a potential and can effectively be used at the victim end for DDoS defense mechanism.

Preface

This thesis is submitted to the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU) for partial fulfillment of the requirements for the MSc. degree in Telematics - Communication Networks and Networked Services. The thesis work has been performed from September 2017- February 2018 under the supervision by Ivar Arnesen from Ivar Arnesen Invest AS and Pof. Yuming Jiang from the the Department of Information Security and Communication.

First of all, I would love to praise God for everything. Secondly, I would like to thank Professor Yuming Jiang and Ivar Arnesen for guiding and motivating me throughout this thesis work. Finally, I would love to thank my family in Ebenezer Church for being there for me during my stay in Trondheim.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

One of the biggest challenges in ensuring network security is detecting and handling of network traffic for possible DDoS attack. DDoS uses multiple systems of a bot-network to generate attack traffic targeting a single victim causing Denial of Service (DoS) of the victim network. It is an attempt to consume finite resources of the victim network in order to make services unavailable to the legitimate users. An attacker builds a bot-network to start DDoS attack so that it can send multiple requests at the same time to exhaust the resources of the victim network. DDoS attack is increasing in size, frequency and complexity. However, nowadays, anyone can launch DDoS attack [Sea]. DDoS attack is a serious threat to companies with an active online business. Therefore, it has become a high priority task to prevent DDoS for the internet stakeholders.

The complexity of DDoS attack makes detection and mitigation difficult. Moreover, it also increases the overall operational costs to deploy mitigation solutions and it is not cost-effective to deploy at the edge of victim networks.

Quite a lot of research has been done to classify DDoS attacks and suggesting techniques to detect and mitigate them [BSA+16],[ZJT13]. Also, there are several open source based intrusion and DDoS detection softwares available online. Open source systems have increased considerable inclination because of their adaptability, support and cost-effectiveness [Sno].

Beside the open source solutions, it has become common and feasible using commodity hardware for network traffic processing [BDKC10]. However, this is not an easy task. It requires careful design and implementation of software to leverage the available commodity hardware resources performance.

Our work is to study the feasibility of using opensource DDoS detection tools in commodity hardware. Furthermore, to set up a complete experimental testbed and evaluate the performance and detection capability of the testbed. We hope this thesis work will help as a benchmark for future works in exploring the potential of software-based DDoS mitigation in commodity hardware.

1.1 Motivation

Some of the driving forces and motivation to pursue the thesis work on this specific topic are:

- Traffic processing in a commodity hardware become a feasible task. The performance improvements made to the software and hardware architecture of commodity hardware brings interest to study and explore it's potential to build network security solutions on it.
- The availability of flexible open source DDoS detection tools and the high cost of commercial DDoS mitigation solutions motivated us to study the feasibility of DDoS detection and mitigation in a commodity hardware.
- Most of the previous research works focuses on detecting DDoS using general purpose network monitoring and analysis Tools, which could be performance bottleneck to detect DDoS in real-time. Therefore, we are interested to implement and experiment DDoS detection by taking performance into consideration.

1.2 Problem statements

This thesis work will try to analyze and give answer to the following questions:

- Can DDoS detection be done in software on commodity hardware?
- What are the processing requirements of such an approach?
- What are the business potential of this approach.

1.3 Objectives

- To evaluate and select the best DDoS detection tool from the existing open source DDoS detection solutions based on an important characteristics.
- Customizing the open-source software and configuring the hardware in order to leverage the available hardware performance.
- To carry out performance analysis in the implemented experimental tested.

1.4 Contributions

This thesis provides the following contributions:

- We built up an experimental testbed composed of a chosen open source DDoS detection tool deployed in a commodity hardware and DDoS attack generator tools.
- We have modified the detection source code and implemented forwarding and filtering function which is essential to deploy in inline mode so that we can evaluate the performance based on different metrics.
- Some scripts are developed that helps to measure the experiment and create filter policy for detection application.
- Finally, the performance and detection accuracy carried out in the tesbed and discussion and conclusion are made based on the obtained results.

1.5 The thesis Outline

- **Chapter 2: Background** -The background chapter will provide the reader general overview of the technologies which is a part of this thesis work. Some of the literature review and related works also summarized in the subsection of this chapter.
- **Chapter 3: Methodology**- Discusses the research methodology used to address the research question of this thesis.
- **Chapter 4: Implementation and Tools -** This chapter presents all necessary information regarding tools and implementation made to complete the experimental testbed.
- **Chapter 5: Evaluation methodology -** This chapter define all metrics and scenarios used for evaluation in the testbed.
- **Chapter 6: Results -** This chapter presents the results obtained from the experimental measurements.
- **Chapter 7: Discussion -** Presents analysis and discussion of the obtained results.
- **Chapter 8: Conclusion -** This chapter includes conclusion made based on the findings of this thesis and some recommendation for future works.

The modified source code of the chosen DDoS detection software and scripts we have developed can be found in the Appendix A,B.

Chapter 2
Background

This background chapter will provide general overview of the technologies covered in the scope of this thesis work. An Introduction to DDoS, classification of DDoS, DDoS detection methods and defence mechanisms as well as challenges and improvements made to commodity hardware for traffic processing will be discussed briefly. Some of the literature review and related works will also be summarized and presented in the subsections of this chapter.

2.1 DDoS

Amongst various types of threats targeting to compromise the security of information assets, DDoS attack targets an availability and utility of computing and network resources. It is an attempt to consume finite resources of the service provider to make services unavailable to the legitimate users. DDoS is an attack where multiple systems (attackers) are used to target a single victim causing Denial of Service DoS of the victim system. The effectiveness of DDoS attack is achieved by using compromised machines and networked devices like botnet as a source of the attack to overwhelm the victim's finite resources. Another approach, which is possible but not common, is an attacker using a single machine by means of generating packets with the spoofed-IP[1] address as the source IP address so that it will arrive at the victim side as it's from different machines or sources. Using spoofed-IP is common in both approaches. The effect of an effective DDoS attack result in service unavailability and has a potential costly consequences.

In the DDoS-attack, attacker commonly uses a bot-network (botnet) also known as a zombie network[NIG$^+$17]. Most of the DDoS attack is generated from multiple compromised machines. An attacker, from a single source, builds bot-network before it starts the DDoS attack. First, an attacker remotely gains access to

[1] Manipulating Internet Protocol (IP) packets with a false source IP address.

compromised machines and makes them part of the bot-network. Botnets are considered as a perfect tool for launching DDoS attacks[RGMFGT13]. Botnets are employed by an attacker to generate multiple requests at the same time to exhaust the resources of the victim as shown in Fig 2.

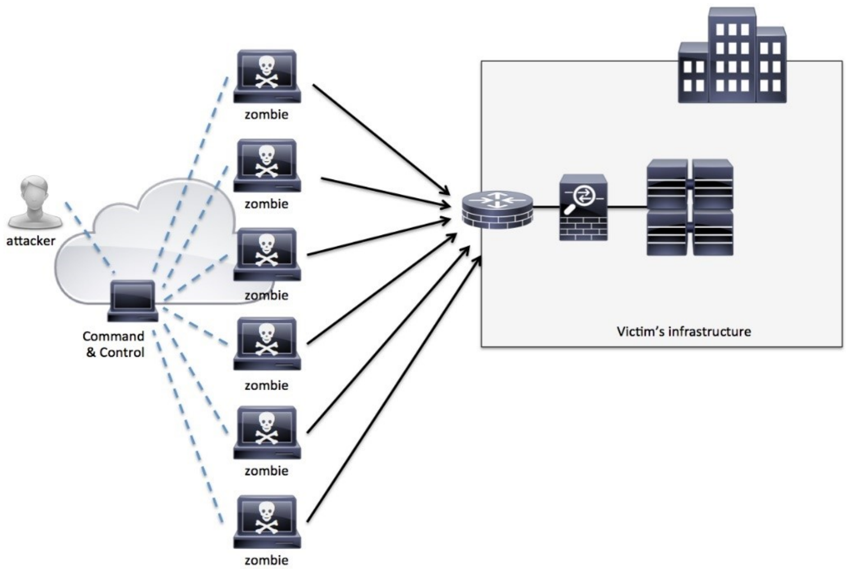


Figure 2.1: Bot or Zombie based DDoS attack [BDKC10]

2.2 Types of DDoS attacks

Several research surveys have been published on the DDoS attack classification and category [BSA+16],[HSS12], [NIG+17], [MR04]. Different DDoS attacks use different approach to attack any part of victim network or services that are vulnerable to attack. Parts of the attack surface could be servers, protocols, routers, applications or databases.

Bhardwaj et al.[BSA+16] study presented a review on literature research work on the DDoS attack on the cloud. They introduced new DDoS attack classification taxonomy based on a different basis. They Classified DDoS on basis of per degrees of automation, vulnerabilities exploited, attack rate dynamics and impact of the attack. The classification of attacks under vulnerabilities exploited includes bandwidth depletion and resource depletion, this includes most of the current DDoS attacks and the scope of our thesis work also focus on attacks under this classification. The bandwidth depletion involves flooding of The Internet Control Message Protocol (ICMP), SYN and and amplification attacks.

Generally, DDoS attacks can be classified into three main groups based on type and magnitude of traffic used: volumetric or volume based attacks, protocol attacks and application attacks. According to [CPPM], DDoS is divided in to two based on the attack target: Infrastructure layer and application layer. In this section we will discuss the infrastructure layer, which includes volumetric and protocol attacks.

2.2.1 Volumetric/ Volume based attacks

This type of attack saturates the bandwidth of the network by sending packet storm. The magnitude of attack is measured in Bits per second (bps). The attack involves bots and zombies to send a huge amount of traffic to exhaust the bandwidth capacity of the network. The effect of the attack saturates the network links and overwhelms routers, switches, firewalls and Internet Service provider (ISP) and overall network level devices. Afterwards, the legitimate users request will be dropeed from reaching to service provider end. Common attacks of this category are UDP flood, TCP flood, ICMP flood and packet flood.

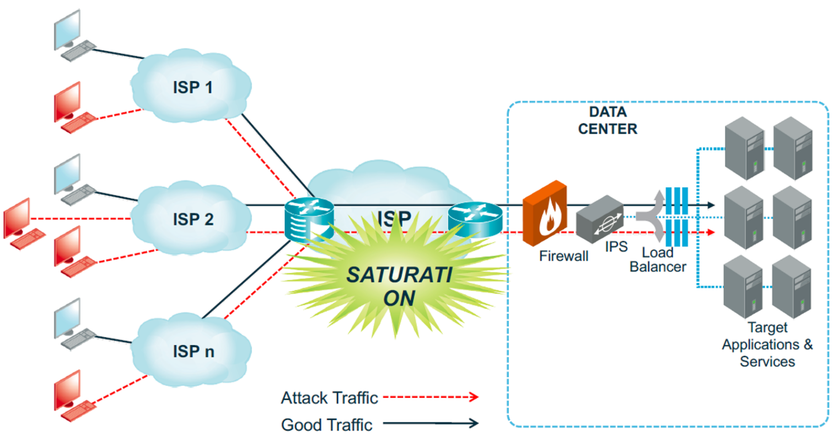


Figure 2.2: Volumetric DDoS attack [Net]

UDP flood attack

The stateless, connection-less communication model, nature of UDP makes a common tool for different attacks which requires manipulating packet. UDP packet is easy to construct and generate. As it is stateless, it is easy to forge source IP so that it could be spoofed and hard to trace the right source of the sender. Therefore flooding using UDP packets become one of the most well known and compelling methods for DoS and DDoS attack [XMZ09]. UDP can be constructed as a very small packet, so that the attacker can easily send a high volume of small-sized UDP packets which causes forwarding issues for network level forwarding devices such as routers, firewalls, and inline traffic processing devices. The less effective UDP flood attack can cause jitter and latency in real time streaming protocols for voice and video.

Under the normal condition, a server which receives UDP request goes through two steps. First, the server checks if a requested port is open and a specific application is running to handle the requests coming through the port. Second, if there is no application is running to handle the request it will respond with ICMP packet setting destination unreachable flag to inform the source address that a unavailability of the requested service. During UDP flood attack, the attacker uses a large flood of UDP with spoofed-IP address and saturates network resources with the request and also with the same amount of destination unreachable ICMP packets responses. As a result, the finite resource of victim network will be exhausted by the process of checking and responding for a huge volume of UDP request floods. This results in denial of service for legitimate traffic.

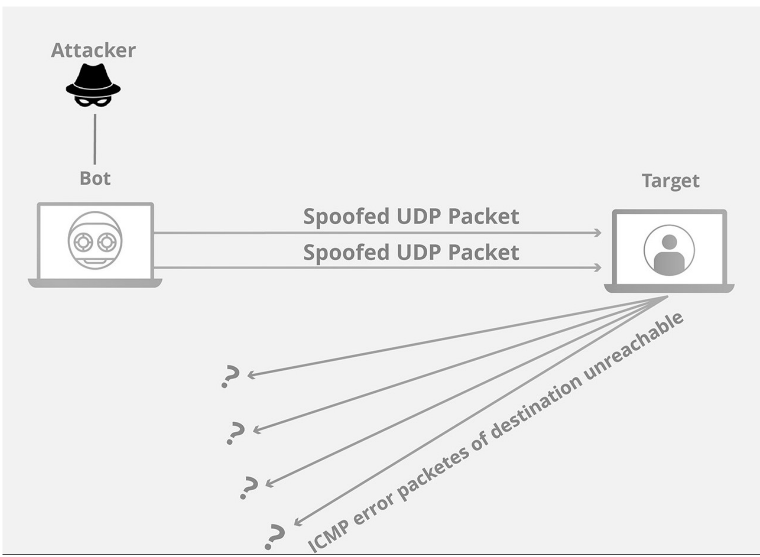


Figure 2.3: UDP flood attack[Inc]

2.2.2 Protocol attacks

Protocol attack works by exploiting a weakness in transport layer and network layer protocols of Open Systems Interconnections (OSI) models based applications and protocols in victim network. It misuses a specific feature or implementation bug of protocols used at the victim network in order to exhaust its limited resources [DM04]. Magnitude of the attack is measured in Packets per second (pps). This type of attack exhausts resources of server and intermediate equipment's working in layer 4 and 5 such as load-balancer and firewalls. Common and well known attacks of this type is Transmission Connection Protocol (TCP) SYN flood.

TCP SYN flood attack

TCP is connection-oriented protocol, unlike UDP. It provides flow control, reliable, ordered and error control services for an application using TCP protocol. Before sending data using TCP it must go three steps known as the TCP three-way handshake to setup a reliable connection [LSBM15]. First, the initiator host sends TCP-SYN (synchronize/start) and then the receiver sends SYN-ACK (synchronize/acknowledge) packet back, Finally the initiator sends ACK. Afterwards, the data communication carries on the established reliable connection.

TCP SYN flood attack uses the first step of TCP three-way handshake stages and sends a huge amount of TCP SYN request to exhaust the victim server. In a normal operation, the server receiving TCP SYN will send back SYN ACK flag and waits for ACK or timeout to expire the connection. Like other DDoS attack, TCP SYN flood attack sends TCP SYN packets from multiple sources with spoofed IP addresses. While trying to handle every request from the attacker which is TCP SYN flood the server become busy and it fails to respond to the legitimate users' requests. Due to the limited resources of server is exhausted by the attack traffic, it creates Denial of Services condition to the legitimate users.

2.3 DDoS Detection methods

According to [AR12], there are two types of network attack detection or intrusion detection methods: signature based detection or anomaly based detection.

2.3.1 Signature detection

Signature based detection, uses a predefined sets of signatures to inspects the network traffic for the presence of attacks [MR04]. The detection application employed this mechanism will compare each packet, commonly it's payload, of the network traffic with a given set of patterns of DDoS attacks. This method is capable of attaining high accuracy and less false positive in identifying attacks. However, it

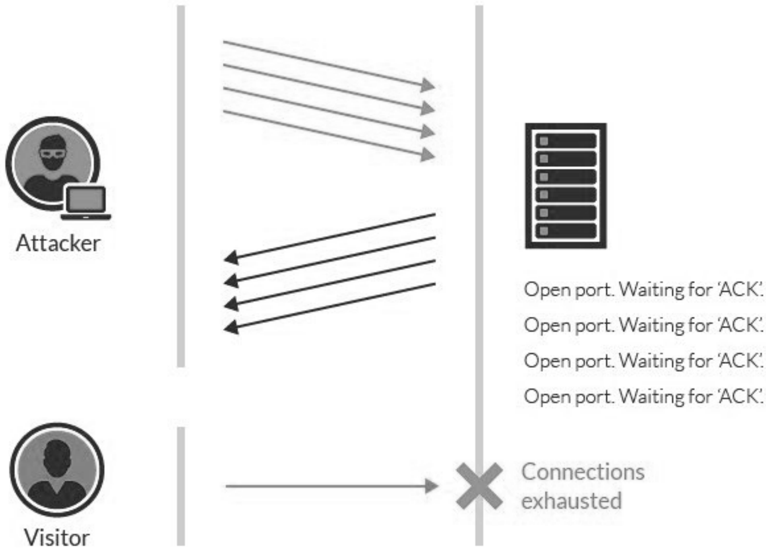


Figure 2.4: TCP SYN flood attack [Imp].

fails to identify unknown attacks which has no stored signature in the a given set of patterns for an attack.

Amtul et al. [SAA13] did experimental evaluation of signature based detection method, Snort [2], against DDoS attack. They have analyzed snort detection capability and accuracy against TCP flooding attack under different hardware configuration. Based on their findings, Even though, signature based detection helps to achieve low false positive, They strongly suggest the need to develop different detection methods like flow-based to analyze packets by checking only the protocol header of incoming packets in the form of flows or groups. They indicate that flow-based DDoS detection could be more efficient and faster than signature based detection as less information is extracted from packets to detect attacks. Finally, they have suggested integrating flow and signature based detection will make it much more proficient by referring to H. Alaidaros et al. [AMAM+11] research work.

2.3.2 Anomaly detection

Anomaly detection unlike signature based it identifies malicious traffic in a network by detecting anomalies network traffic pattern [HAG10]. The behavior of network can be analyze in different ways, for example:

[2] https://www.snort.org/

– Analyzing using packet size to check if the size is too short and violate application layer protocols.

– Rate-based detection uses a time-based profile of normal traffic volume to detect against DDoS flooding attacks.

The advantages of the anomaly detection over signature based is that it is not limited to known attacks, it can detect previously unknown attacks based on the behavior of the attack traffic.

S.H.C. Haris et al. [HAG10] did research on how to anomaly detect TCP SYN flood attack. They have developed algorithm to detect TCP SYN attack by analyzing Internet Protocol (IP) and TCP protocol header. They have presented the experimental result and found that their algorithm based on anomaly detection method can detect TCP SYN flood in the network.

2.4 DDoS defense mechanism

Along with the increasing DDoS attack in size and complexity, many research have been done to propose defense mechanism and classification based on different basis. DDoS defense mechanism can be classified into three based on their deployment locations [BSA+16]: victim-end, source-end and intermediate router defense mechanism, as shown in Figure 2.5.

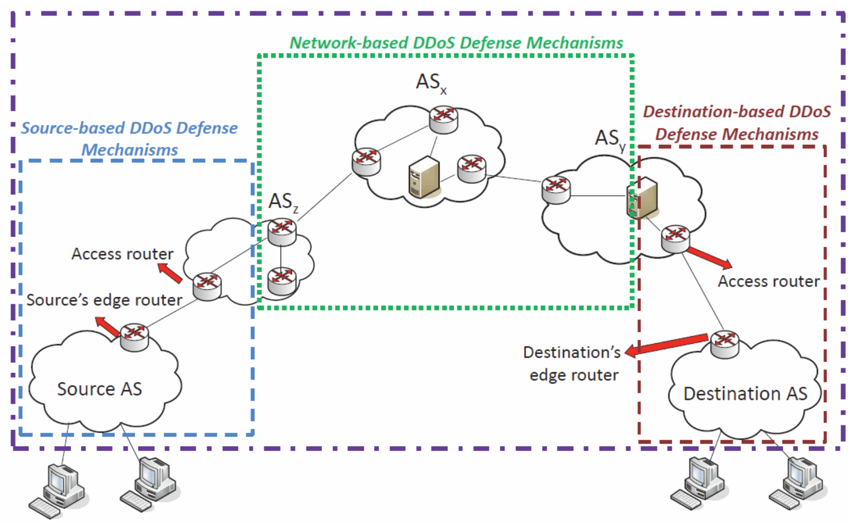


Figure 2.5: Deployment locations of DDoS defense mechanism [ZJT13]

2.4.1 Victim-end defense mechanism

Victim-end defense mechanism is deployed at the gateway of the victim network, mostly employed in the edge router of the victim network. DDoS attack detection at the victim end is comparatively easier than others, the collectively high volume of incoming traffic from distributed attackers can be used as a sign of DDoS attack [MPR03]. Currently, most defense systems are located at the victim end and also the most motivated to deploy DDoS defense [MR04].

While detecting DDoS using victim-end mechanism employed in the router of the victim network is relatively easy, the problem is that the victim resources including the router could be overwhelmed by DDoS attack and the legitimate users request could be denied. Therefore, effort should be made to minimize the computation required for attack detection while maintaining high detection accuracy [KB12].

2.4.2 Source-end defense mechanism

It has the same architecture to victim-end defense mechanism. It is deployed at edge router to prevent the source from generating DDoS attack. The outgoing traffic is monitored and filtered by DDoS detection and mitigation solutions. Source-end detection can be achieved by deploying solutions such as: source-end firewall using signature based detection, threshold anomaly detection by defining a set of thresholds for various traffic types such as average packet rate per connection and average number of outgoing UDP packets per destination [MPR03]. However, detecting DDoS at source-end is not easy because of the widely distributed sources of DDoS attack [SAA13].

2.4.3 Intermediate or Network-based DDoS Defense Mechanisms

Network-based DDoS defense mechanisms deployed in intermediate routers of the autonomous systems. Routers exchanges information on detected source of generating DDoS attack. Some of the main network-based DDoS defense mechanisms presented in [ZJT13] are: route-based packet filtering and detecting and filtering malicious routers. This mechanism is not yet used widely, because it is not considered as effective and efficient because of their large overhead of network communication [ZJT13].

2.5 Traffic processing in Commodity Hardware

Nowadays, deploying network security solutions are based on a specialized equipment for specific solutions from different vendors such as: Firewalls, load-balancer, intrusion detection and prevention system to control abnormal traffic. The high cost of Distributed Denial of Service (DDoS) mitigation devices is not a cost-effective model to deploy them at the true edge of the victim network [CDL16]. On the other hand, commodity hardware are available, which are capable of handling big traffic in Giga speed and equipped with high multi-core processing power. Capturing network traffic with commodity hardware has become common in many industries, improvements in modern commodity hardware and software architecture exhibit tremendous power which previously were the domain of expensive special purpose hardware [BDKC10].

Commodity hardware and its subsystems are designed for general purpose usage. While this approach is enough for general purposes, it is a performance bottleneck for applications processing high-speed traffic on commodity hardware. The packet journey from Network Interface Card (NIC) to the processing application is long because of its initial design purpose. Another challenge is, the constant increase in internet link speed and attack magnitude brings addition performance overhead to implement software-based traffic processing application in a commodity hardware.

Processing traffic in such hardware has to be quick and low-latency. In order to process and forward network traffic in high-speed, packet capturing and forwarding has to be fast possibly operate in line-rate[3]. Several researches have been done to improve the packet capturing subsystem of the commodity hardware [BDKC10]. Another performance bottleneck is packet delay in a processing application. Every packet received has to be examined and forwarded by the software. If the software is not designed to minimize packet delay while processing, it could be another limitation to the performance.

2.6 Life cycle of packet in Commodity hardware

In a commodity hardware, capturing packets involves several software subsystems as illustrated in Figure 2.6. NIC manages incoming and outgoing packets by copying to kernel space and sending out from kernel space. The driver is notified when new packets arrive and when there is a packet to be transmitted, the driver copies to NIC and notifies by updating a card register in NIC. The kernel thread responsible for packet handling copies packets to network stack of the operating system. Finally, packets are provided to packet processing application by user-space packet capturing libraries like libpcap [4].

This long process of traditional packet capturing and forwarding solution in commodity hardware limits the packet processing performance. It incur a large amount

[3] is the maximum capacity to send frames of a specific size at the transmit clock frequency of the Device Under Test [For]

[4] a portable library for network traffic capture.

of overhead as operating system copies packets multiple times. It makes unable keep up with high-speed traffic, resulting in packet drops. While costly special purpose commercial hardware exists to handle high speed traffics, different software-based approaches introduced, which improved the capturing and forwarding performance in commodity hardware. Some of the frameworks are presented in this section.

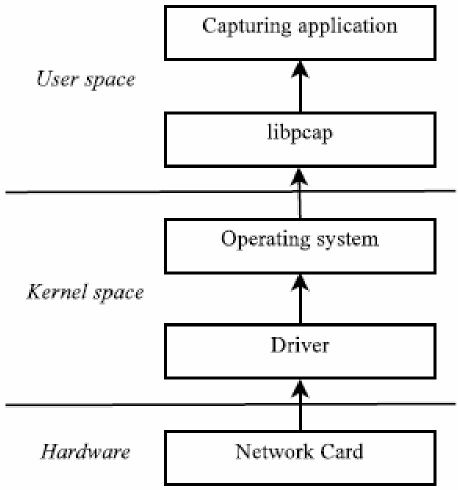


Figure 2.6: Traditional packet capturing process in commodity hardware [BDKC10]

2.6.1 Netmap

Netmap is an open source software framework uses kernel module and modified NIC drivers in order to improve the speed of packet capturing and forwarding in a commodity hardware. [Riz12] defines netmap, *a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.* NIC using Netmap can be operate in two ways: regular mode where the NIC exchanges packet with the host stack as usual and netmap mode, where the the data path is disconnected between the NIC and the operating systm [RDC12]. Netmap provides fast access to network packets API for traffic processing applications in user space, as shown in Figure 2.7. This framework provides a huge performance improvements to a wide range of applications require fast packet capturing and forwarding such as: software based routers and firewalls. [Riz12].

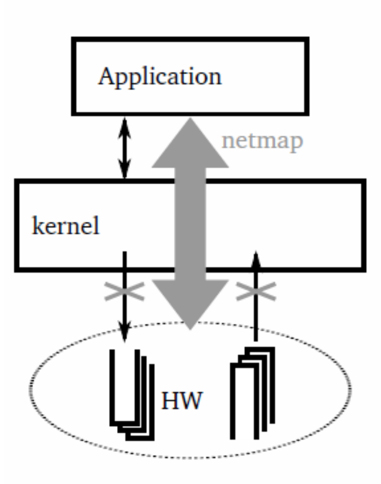


Figure 2.7: Netmap based packet capturing [BDKC10]

2.6.2 PF_RING

PF_RING is another fast packet capturing framework developed by Ntop [nLc]. PF_Ring also disconnects kernel intervention in packet capturing and forwarding process. It implements a memory-mapped memory buffer (socket ring) where the incoming packets are copied and user-space applications can simply access this memory, as shown in Figure 2.8. Ntop introduced variant version of PF_Ring with some advanced features. An open source version called Vanilla and commercial version Zero Copy (ZC) [nLc].

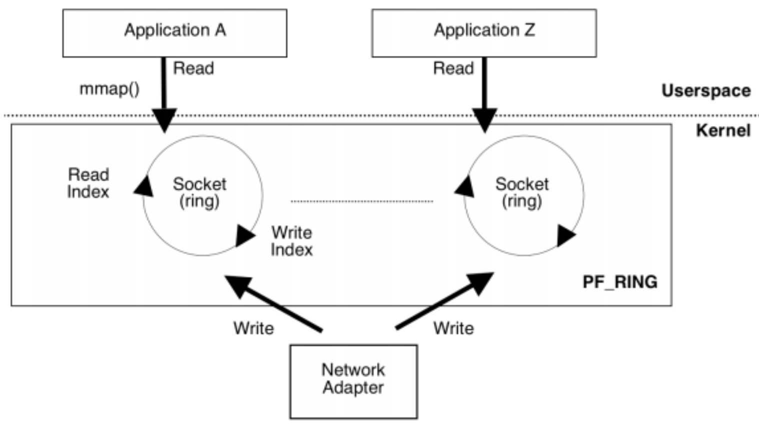


Figure 2.8: PF-Ring based packet capturing [D+04]

2.7 Related works

Alfredo et al. [CDL16] presented the design and implementation of a software based DDoS mitigation called nscrub [nLa]. They have tested and validated the detection performance and accuracy in their testbed using commodity server and traffic generating tools. The result of their research has shown the practicality and feasibility of using DDoS detection software in commodity server for edge network traffic up to 10Gb/s. nScrub uses signature based packet inspection method to detect DDoS attack. However, using only this method for DDoS attack is inefficient as stated in [PLR07]. And also we were interested to include nScrub as candidate tool but they have only commercial version.

Route Soumaa [Mer17] published a paper titled "An Approach for Detecting and Preventing DDoS Attacks in Campus". They have evaluated Snort , open-source Intrusion Detection System (IDS) in terms of packet processing and detection on windows server 2012 with XEON processor and 128 GB ram. They have installed Snort with winpcap, windows version of libpcap PCE. On thier experiment LOIC tool is used to simulate botnet based DDoS flooding attack. They have defined different metrics to evaluate the effect of attack on victim resources (Central Processing Unit (CPU) load and Memory). They have improved Snort detection capabilities in terms of accuracy by modifying the available rule sets, which is based on signature based detection method, to protect DDoS attack. Overall, their work show possibilities of detecting and preventing DDoS using open-source solutions and to suggest new approach for Snort campus network security solutions. Their proposal is to show the possibilities of detecting DDoS using snort but they did not discuss or explain the limitation of such approach in high speed network traffic and attack.

Jati et al. [JHP+16] also did research in similar topic to detect DDoS using open-source traffic monitoring tool, Ntopng[nLb], on hardware equipped with 2 Intel processor speed of 1.8 Ghz , 4GB memory and link capacity of 2Gbits/s. They evaluated Ntopng in terms of accuracy , sensitivity and resource usage. The result shows that the maximum traffic handling capacity of Ntopng in a given hardware configuration is about 128 Mega bits per second (mbps). Ntop uses libpcap and signature based detection method and it is designed for general network traffic monitoring purpose. Using such approachs can be performance bottleneck to tackle high volume DDoS like volumetric attacks.

2.7.1 Summary

Most of the proposed works and researches we have covered in the related works mainly focus on possibilities of detecting DDoS using open-source tools, which are designed for signature-based IDS or general traffic processing or monitoring. And also, their proposed architecture and design did not consider or included to improve commodity hardware limitation on packet processing, except Alfredo et al. [CDL16] design. Since our objective is to evaluate the performance and accuracy of open source based DDoS detection in commodity hardware, these issues have been taken into consideration during the selection process of DDoS detection tool and using commodity hardware.

Chapter 3
Methodology

In order to address the research question stated as problem statement, we have mainly designed and implemented an experimental tesbed and some additional steps are carried out to supplement that. Using the testebed, we conducted experiments to evaluate DDoS detection system performance and detection accuracy while detecting DDoS attacks. Figure 3.1 shows the logic and steps of the work-flow and the overall steps. The methodology used in this thesis work includes the following ones:

Literature survey: A literature survey was first conducted in different online databases such as Google scholar, Association for Computing Machinery (ACM) , springer and Institute of Electrical and Electronics Engineers (IEEE) using a key-words: DDoS detection, anomaly detection, DDoS mitigation and traffic processing in commodity hardware. We have reviewed different research works related to our topic and found suggestions to appropriate tools and methods for our experiment.

Implementing the missing feature: In direct follow up to the literature survey and before setting up the experimental testbed, we have chosen fastnetmon open-source DDoS detection application based on the important characteristics such as detection methods and type of attacks it support. Afterwards, we have identified the missing feature and implemented to complete the functional requirement for our testbed. The selection criteria and implemented feature is covered in chapter 4

Experimental testbed setup: The chosen detection application with our implemented feature is deployed in to our testbed hardware. Different tools that would be essential for this study which are selected based on the previous experiences of different research works are also installed in the testbed hardware. Finally, different performance and accuracy metrics are defined to carry out evaluation and measurements.

Measurements and analyzing results: the results from measurements are analyzed, discussed and conclusion and future works are made.

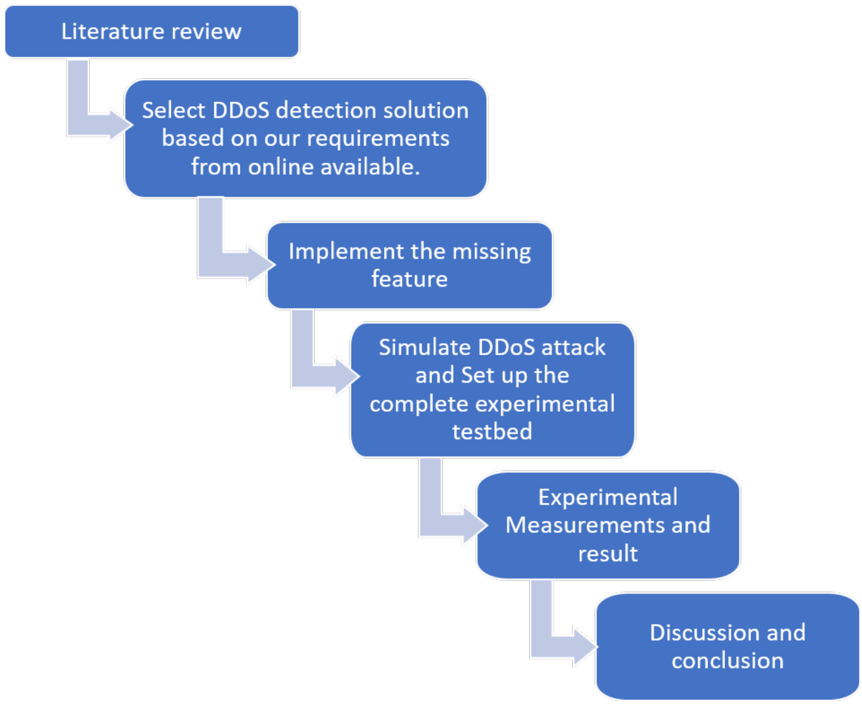


Figure 3.1: Logic of the workflow used to address the research question

Chapter 4

Implementation and Tools

This chapter presents the different tools and implementation carried out to setup the experimental testbed. The important features of the chosen DDoS detection software, added feature and necessary information regarding the tools used in our testbed are described.
When setting the experimental testbed, we relied on the design and requirements developed by the author in the study [Der03]. The author defines the decisions he made based on the objectives of his research. The decisions we have made to conduct our research are:

– All the hardware components need to be available on the market at reasonable price.

– All the softwares used need to be open source and available for free.

– The softwares need to be flexible to modify in order to leverage the available hardware capabilities.

4.1 Choice of DDoS detection software

Our experiment is based on commodity hardware and open source software. On the software side, the requirement to use open source software is motivated by the availability, flexibility and it's practicality of providing solution for network security based on the previous works. It is very important to have an essential requirement to choose the best open source from those available on the internet. The requirements are based on the important characteristics of detection engine that which are to be evaluated in the implementation. The following lists of requirements have been taken into consideration during the selection process of DDoS detection tool from online available software:

- The capability of detecting the attack categories described in the previous chapter.
- Detection method or algorithm used.
- The flexibility of the software to modify.

The list of candidate DDoS detection open source tools are presented in Table 4.1. After validating the listed DDoS detection tools, FastNetMon is found to be the best open source DDoS detection software based on our requirements. Most of the open sources we have validated doesn't support detecting the well-known DDoS attacks, where as FastNetMon detects most of the infrastructure layer DDoS attacks targeting network level devices. FastNetMon has two version: Community version, which is open to everyone and with limited detection capability, while advanced or commercial version supports advanced detection and mitigation features[LTDb]. The community version we have used has the necessary features required to implement in our experimental testbed. FastNetMon also has many online forums and well-organized documentation which can simplify the installation and customization process.

4.2 FastNetMon

FastNetMon is a very high-performance DDoS detector built on top of multiple packet capture engines: PF_Ring, netmap, sFLOW, Netflow, PCAP. One of the interesting features of FastNetMon is that it supports most of the network vendors and has a flexibility to be installed and modified by developer in different Linux distribution including Debian, CentOS, Ubuntu, Fedora and Gentoo. As it is designed to detect DDoS attacks, it has core algorithms that detect a pattern of different DDoS attacks. It supports anomaly detection using rate-based and protocol based to the hosts in the network. It also has additional signature-based deep packet inspection (DPI) against false positive attack detection.

Example of FastNetMon deployment scheme is presented in the Figure 4.1. Figure 4.2 presents traffic flow in FastNetMon and it's main software components. The main FastNetMon software components are: Policy manager, PCE, detection engine and report manager. The policy manager is responsible for selecting one of the packets capturing modules and initializing resources (memory and CPU) based on the given hardware configuration preferences. Detection engine analysis every packet passed by the selected PCE. For some attacks, if the selected PCE provides packets with payload then advanced DPI will process the packet for false positive attack detection. Finally, report manager reports based on the detection status whether

Table 4.1: Open-source DDoS detection tools

Detection application	Description
FastNetMon	High performance DDoS mitigation tool which is based on a packet analyzer engine (PF_RING, netmap, sFLOW, Netflow, PCAP) [LTDb].
Snort	It is an intrusion prevention system capable of real-time traffic analysis and packet logging based libpcap packet capturing engine[Sno].
ntopng	High-Speed Web-based Traffic Analysis and Flow Collection [nLb].

the incoming traffic is an attack or not. Afterwards, different policy enforcement devices may take an action based on the report.

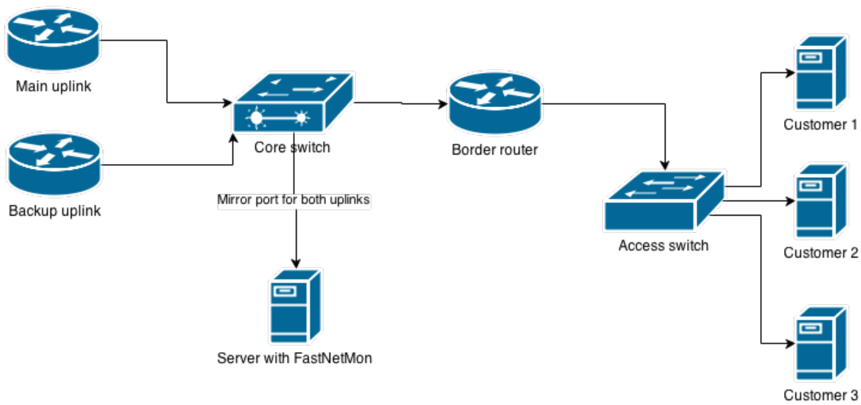
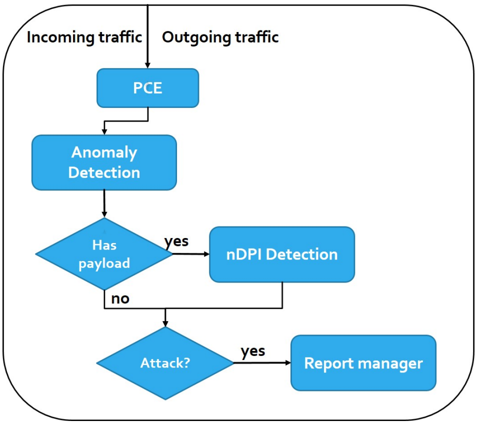


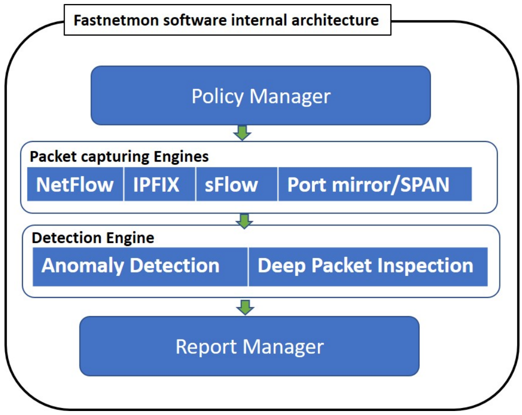
Figure 4.1: FastNetMon deployment scheme [LTDa]

4.2.1 Fastnetmon PCE

Network traffic analysis is a process used to monitor the communication pattern between hosts and towards internet in the network. This involves capturing traffic which may give limited information of a packet, which is a flow data or detailed information including packet payload. Fastnetmon supports most of current packet capturing techniques and frameworks. It supports NetFlow, sFlow and IPFIX based flow data analysis for traffic collected from devices such as router or switches. This data commonly used to track key fields like: source interface, source and destination IP address, layer 4 protocols, source and destination port numbers and type of service value.



(a) Flow diagram of detection in Fastnetmon



(b) FastNetMon software architecture

Figure 4.2: Fastnetmon internal architecture

Fastnetmon also supports high performance packet capturing frameworks discussed in previous chapter such as netmap and PF_Ring ZC as well as common but slow packet capturing library libpcap. There is netmap-enabled version of libpcap, which enables libpcap based applications to run on top of netmap at much higher speeds. These frameworks provide packets with payload , so that FastNetMon can apply deep packet inspection on the packet of the network traffic.

4.2.2 FastNetMon Detection Method

Fastetmon detection logic is based on both anomaly and signature based detection methods, As can be seen from Figure 4.2. Anomaly, it detects based on the rate of the traffic incoming to or outgoing from a given networks in Classless Inter-Domain Routing (CIDR) format by policy manager. The rate is based on number of pps, mbps and flows per host. For advanced detection if the PCE provides packets with payload it uses signature based detection called nDPI [1][Nto].

Memory consumption of FastNetMon during detection is depends on the total number of monitored hosts. It assigns small amount of memory per host, which are data counter, current speed counter and traffic counters. For the version we have installed, which is 1.1.3, for hosts in /16 network the total memory consumption is about 40 mega byte of a given RAM.

Using the above detection methods, FastNetMon detects the following attack types:

[1]Open and Extensible LGPLv3 Deep Packet Inspection Library

- TCP-SYN flood: TCP packets with enabled SYN flag.

- UDP flood: flood with UDP packets.

- ICMP flood: flood with ICMP packets.

- IP fragmentation flood: IP packets with MF[2] flag set or with non zero fragment offset.

4.2.3 FastNetMon report

After detecting attack FastNetMon report module will write details of the attack in file or dumps traces in pcap for the attack traffics. If FastNetMon is configured to take action based on the report it runs external triggers to :

- notify attack summery using custom script.

- Announce with Border Gateway Protocol (BGP) (EaxBGP) [fas].

The FastNetMon sample configuration file is presented in 4.1.

4.3 Implemented feature

As presented in Figure 4.1 FastNetMon deployment is offline. To have a complete experimental testbed of our interest we have implemented packet forwarding and filtering module by modifying FastNetMon source code. The implemented feature enables us to have inline deployment scheme as displayed in Figure 4.3. Using an inline or transparent deployment, fastnnetmon can operate inline by checking the incoming traffic content and makes decision to forward or discard based on detection result. The modified sub internal architecture of FastNetMon is displayed in Figure 4.4.

[2]More Fragments flag

Algorithm 4.1 FastNetMon sample configuration

```
# We could disable processing for certain direction of traffic
process_incoming_traffic = on
process_outgoing_traffic = off
# Different approaches to attack detection
ban_for_pps = on
ban_for_bandwidth = off
ban_for_flows = off
# Limits for Dos/DDoS attacks
threshold_pps = 28000
threshold_mbps = 3400
threshold_flows = 28000

### Traffic capture methods
mirror_netmap = on
pcap = off
netflow = off
sflow = off
enable_pf_ring_zc_mode = on
interfaces = enp0s25
collect_attack_pcap_dumps = on
process_pcap_attack_dumps_with_dpi = on
# This script executed for ban, unban and atatck detailes collection
notify_script_path = /usr/local/bin/notify_about_attack.sh
# ExaBGP could announce blocked IPs with BGP protocol
exabgp = off
exabgp_command_pipe = /var/run/exabgp.cmd
exabgp_community = 65001:666
exabgp_next_hop = 10.0.3.114
```

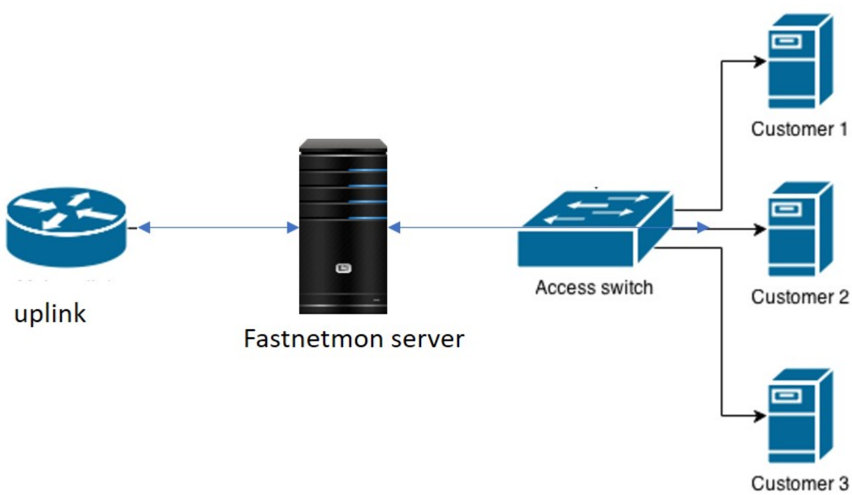


Figure 4.3: FastNetMon inline deployment

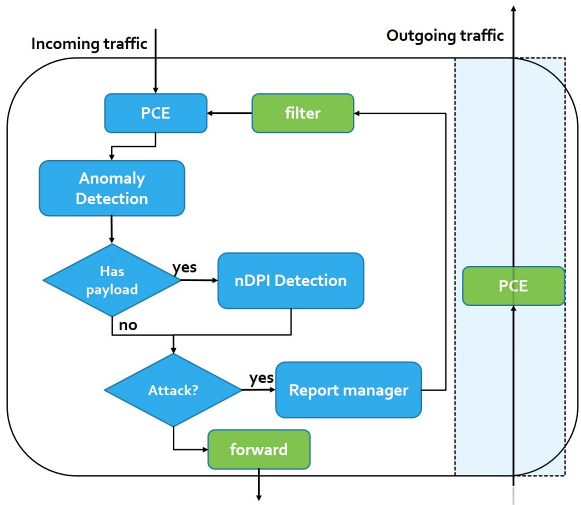


Figure 4.4: Flow diagram of traffic in the modified FastNetMon

We have modified both netmap and libpcap PCE plugins. FastNetMon in offline mode captures both outgoing and incoming traffic from one or more interfaces based on the configuration and detects attack and then writes attack report to files, as depicted in Figure 4.2a. This process is modified to capture packet, detect and forward for the incoming traffic. For the outgoing traffic, which is not being detected, we have developed a script which runs in background. This script bridges the interfaces in one direction, outgoing. This way we could deploy FastNetMon in inline mode and measure throughput and detection accuracy. Both the background and modified source code for netmap and pcap can be found in the Appendix B.

The traffic capturing code of FastNetMon is modified to filter the incoming traffic based on the given rule, which is generated from the attack report file of FastNetMon. The source code snippet in 4.2 shows an example, how the filtering algorithm filters TCP SYN attack reported.

4.4 Tools for packet generation

A lot of tools are available for generating packets in different formats and volumes. The following tools are selected based on our requirements to generate background traffic which are used to simulate traffic under normal operation of the network. Therefore, the selection is based on specially tools with monitoring the exchanged packets in terms of size, type and magnitude as well as based on the previous works experiences.

Algorithm 4.2 source code snippet for traffic filtering.

```
char filterin_expression[255] = " not (dst host 10.10.10.172
            and dst port 2323 and tcp[tcpflags]==tcp-syn) ";
struct bpf_program filter;
bpf_u_int32 subnet_mask, ip;
if (pcap_compile(descr, &filter, filterin_expression, 0, ip) == -1)
{
    printf("Bad filter - %s\n", pcap_geterr(descr));
}
if (pcap_setfilter(descr, &filter) == -1)
{
    printf("Error setting filter - %s\n", pcap_geterr(descr));

}
```

4.4.1 iPerf3

iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks[ipe]. It is a client-server based tool for both UDP and TCP protocols.

4.4.2 pkt-gen

Pkt-gen is another packet sender and receiver application at high rates based on netmap PCE. It is possible to generate packets with a number of tuning options such as: packet rate in pps, packet size and protocol types.

4.5 DDoS attack tools

In our experimental testbed, the most common DDoS tools such as LOIC and hping3 [Hpi] are used based on the capability of types of DDoS attacks they can generate and previous experiences in [SAA13], [NSCP15] [Mer17],[DHKB16].

4.5.1 hping3

Hping3 is one of the de-facto tools for security auditing and testing of firewalls and networks, and was used to exploit the Idle Scan scanning technique now implemented in the Nmap port scanner [LTDa]. Hping is designed to generate packets and analyses TCP/IP protocols. It is a command-line oriented with desirable parameters including:

– flood: sending packets as fast as possible.

- S: TCP with SYN flag.
- D: data size.
- c: packet count.
- Random-source: random the source address or spoofing.

and much more parameters can be passed to hping3. It is easy to manipulate packets using hping, which makes it a best tool for DDoS attack.

4.5.2 LOIC

LOIC is another opensource tool designed to generate common DDoS attacks such as TCP flood, UDP flood, and HTTP flood to a specified web server or IP address. It can simulate bot-network for a given number of hosts, as illustrated in Figure 4.5.

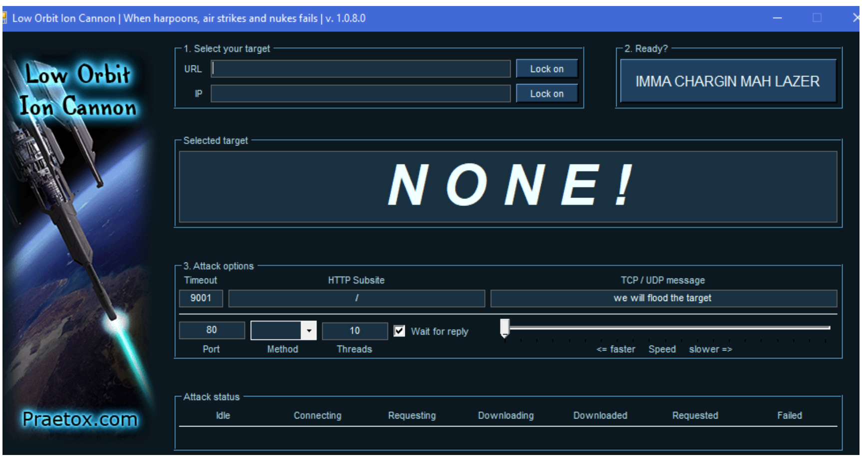


Figure 4.5: LOIC in action

4.6 Experimental testbed setup

The experimental setup of our testbed is a simulation of architecture of victim end defense mechanism, as presented in Figure 4.6. FastNetMon is installed in the hardware with two network interface cards for the incoming and outgoing traffic of the victim network. The victim network hosted simple web-server in the victim server and DDoS attack tools are installed in DDoS attack generator system. Background traffic generator tools are installed in normal traffic generator system.

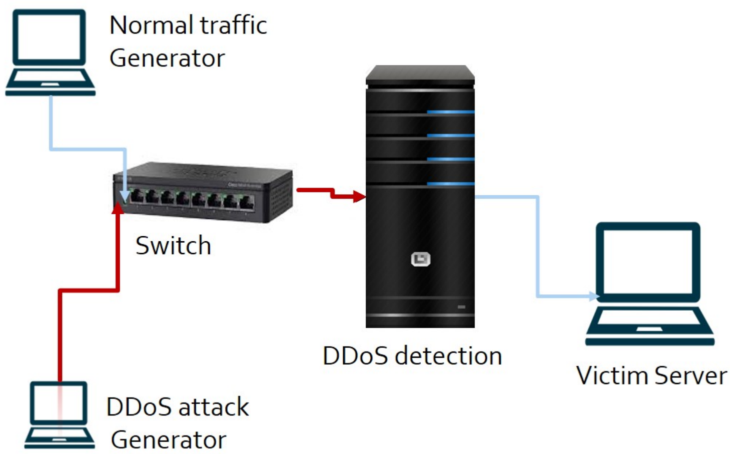


Figure 4.6: The experimental testbed structure

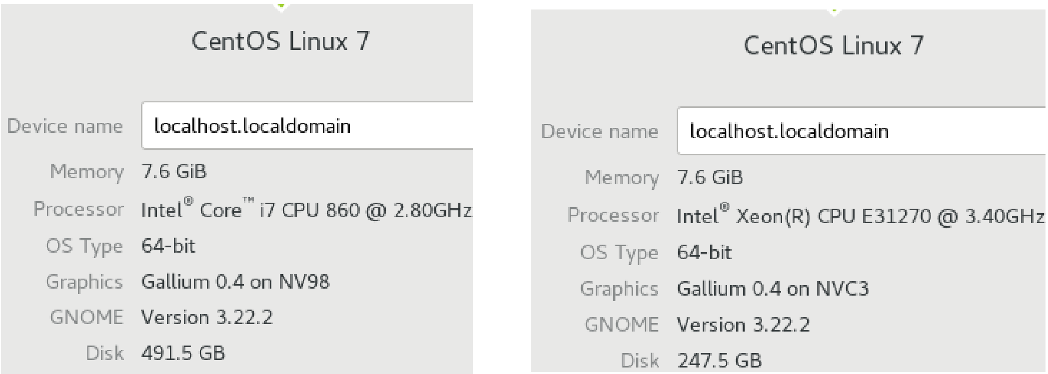
4.7 Hardware

We have used a hardware to experiment on a system equipped with an Intel(R) Core(TM) i7 CPU CPU at 2.80GHz, 8 GB RAM and a dual port 1 Gbit/s card based on Intel NIC the detail hardware specifications for each system is displayed in Table 4.2.

Table 4.2: Hardware specifications

No.	Hostnam	Processor	RAM	OS	NIC
1	DDoS detection	8 Intel(R) Core(TM)i7 CPU 860 @2.80GHz	8 GB	Cetos 7 x64	Dual Intel NIC 1Gibts/s
1	DDoS Generator with Hping	Intel(R) Core(TM)i7 CPU 860 @2.80GHz	8 GB	Kali Linux 7 x64	Intel NIC 1Gbits/s
1	Normal traffic Generator with pkt-gen and iperf3	Intel(R) Core(TM)i7 CPU 860 @2.80GHz	8 GB	Cetos 7 x64	Intel NIC 1Gbits/s

Most of the experimental measurements are done using the above hardware specification. For some measurements, we have changed the DDoS detection processor capacity from 2.80Ghz to 3.40Ghz to compare the performance of FastNetMon in different CPU speed in the same test environment, as shown in Figure 4.7 .



(a) system with CPU @2.80 (b) system with CPU @3.40

Figure 4.7: Systems detail with different processor

4.7.1 FastNetMon in DDoS detection hardware

The chosen detection software, FastNetMon is automatically installed using the following command from git repository:

```
 wget https://raw.githubusercontent.com/pavel-odintsov/
fastnetmon/master/src/fastnetmon_install.pl -Ofastnetmon_install.pl
sudo perl fastnetmon_install.pl
```

In order to modify the source code of FastNetMon, we have installed the community developer version using the following command after the above automatic installation:

```
cd /usr/src/fastnetmon
git checkout master
cd src/build
cmake ..
make
./fastnetmon
```

The configuration file to define being detected network, specifying interfaces and to configure detailed detection preference can be found in /etc/fastnetmon.conf.

Chapter 5
Evaluation Methodology

This chapter will define all metrics used in our experiment and describes the experimental tested in details for different scenarios. The main objective of this thesis work is to analyze DDoS detection on commodity hardware in terms of performance and detection accuracy. For evaluation methodology we relied on the testing methodology used in [PZC+96]. The paper presents detailed procedures for testing an intrusion detection system. We have applied some of the procedures listed such as:

- Intrusion identification test: this method is used to test the detection accuracy of the chosen DDoS detection software.
- Resource usage test: this testing method is used to measure the resource usage of the detection software in different defined scenarios.

5.1 Network topology

The network topology for the experimental testbed illustrated in Figure 5.1. All links in the topology share the same link capabilities and properties. The IP addresses are used to identify the source and destination of the network traffic.

5.2 Evaluation metrics

The evaluation metrics are chosen which can give us a reasonable results for the detection accuracy and performance of fastnetmon in different hardware configuration and scenarios.

5.2.1 Maximum throughput

This metric is used to measure the maximum processing and forwarding capability of the DDoS detection in a given hardware configuration. The processing capability

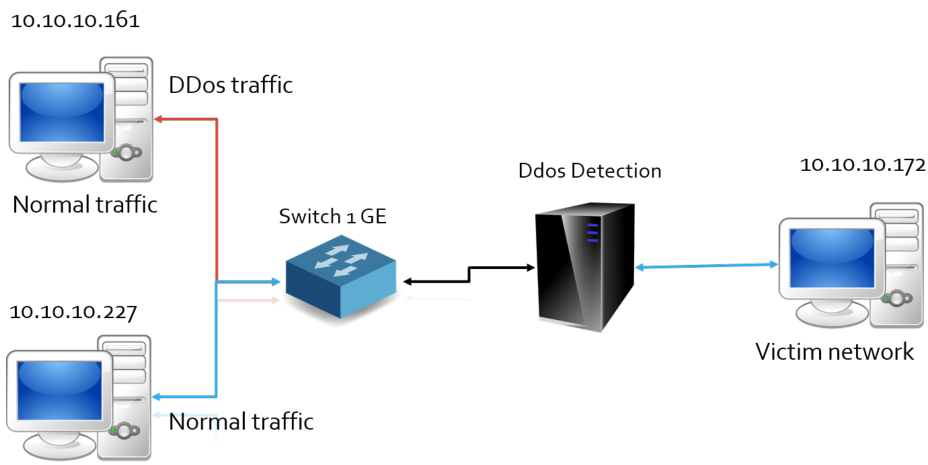


Figure 5.1: Network topology in testbed

is measured in terms of magnitude of pps and bps processed and forwarded. The hardware configurations are carefully assigned and measurements are recorded using different tools for a reasonable period of time. Maximum throughput can be calculated by measuring the input and output traffic in a given hardware configuration. For throughput test we followed the guidelines for throughput test presented in [tB17], which the author listed out from Request for Comments (RFC)s as standard and best practices.

5.2.2 Packet delay

When processing packet in inline packet processing applications, the latency should be low and the processes must be quick. The packet delay is measured to study the packet processing time of the detection software. The ping utility is used to measure the round-trip delay time with and without running the detection application. The packet delay is calculated from the measured results.

5.2.3 Resource usage

The resource utilization(CPU and RAM) of the detection software in different scenarios and hardware configurations are measured and analyzed. Resource utilization is measured using resource monitoring tools selected for this experiment.

5.2.4 Detection accuracy

The detection accuracy of the software is measured to study the attack detection accuracy of fastnetmon following the intrusion identification test procedure and calculated using the equation in [KS11]. The parameters used are:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5.1}$$

- True positive: number of packets correctly predicted as attack packets.
- False positive: number of packets incorrectly predicted as attack packets.
- True Negative: number of packets correctly predicted as normal packets.
- False Negative: number of packets incorrectly predicted as normal packets.

5.3 Scenarios

We have selected three scenarios with in the same hardware configuration: under normal operation, under UDP flood attack, and under TCP SYN attack. All of the scenarios are carried out in the same hardware configurations and measured and logged for analysis.

Hardware configurations used for the scenarios are summarized in the table Table 5.1. Beside the link speed and capacity limitation of the hardware in our testbed, the reason we have used a single core CPU is that fastnetmon shares the traffic load to the CPUs using load-balancer based on the IP addresses of victim network hosts.

Table 5.1: Hardware configuration for the scenarios

No.	Processor	RAM	NIC speed
1	1 * Intel Core(TM) 860 @2.80 GHz	8 Giga Byte	a dual 1 Gbits/s

5.3.1 Under normal operation

Network normal operation is simulated by generating network traffic classified as normal. Fastnetmon detection accuracy and performance are examined by generating a variable amount of normal traffic. As shown in page 34 the normal traffic generator from 10.10.10.161 sends normal traffic to the victim server 10.10.10.172. The fastnetmon resource usage and the above-mentioned metrics have been analyzed.

5.3.2 Under UDP flood attack

UDP flood attack is generated from DDoS attack generator, 10.10.10.161 with different traffic magnitudes. Normal traffic is also mixed which is generated from normal traffic generator, 10.10.10.227. The detection accuracy and performance of fastnetmon have been analyzed. This scenario is used to study the effect of volumetric attack on both the fastnetmon and resources of victim network.

5.3.3 Under TCP SYN flood attack

TCP SYN flood is generated from 10.10.10.161 with mixed normal traffic from 10.10.10.227 to the victim system. TCP SYN flood effect on fastnetmon detection software and victim network resources are measured and analyzed.

5.4 Measurement setups and tools

The tools used for measurements are summarized in Table 5.2. The resource usage test procedure used in [PZC$^+$96] and we have followed are:

- Minimize other background activities and process in the test environment.
- Start the detection software.
- Start testing scripts.
- Start tools to measure and monitor resources.
- Repeat and save the logs.

Table 5.2: Tools for experiment measurements

Tools	Description
taskset[Lov]	is used to set or retrieve the CPU affinity of a running process given its process-id or to launch a new command with a given *CPU affinity*.
htop[hto]	process viewer for Linux used to measure resource usage.
nload[Rie]	displays the current network usage in realtime.
tshark[Wir]	Dump and analyze network traffic.
ping[Wir]	measure delay.

5.4.1 Starting DDoS detection

The command to start FastNetMon DDoS detection in a given hardware configuration Table 5.1, which is a single CPU core is:

```
taskset 0x0 ./fastnetmon
```

The above command will bind the fastnetmon process to the specific CPU ID 0.

The following command is used to capture and dump the incoming traffic to the victim network and also bind the process to other available CPUs .The traffic is filtered by src MAC address of the generator in order to identify the amount of traffic received at the detection system. The command is:

```
taskset 0x4 tshark -i enp0s25 ether src  00:0c:29:a5:ce:fa -F
pcap -w attacker.pcap
taskset 0x5 tshark -i enp0s25 ether src  18:03:73:ad:ce:ba -F
pcap -w normal.pcap
```

we have used the physical address instead IP because DDoS attack tool generates a traffic with spoofed-IP sources.

5.4.2 Starting DDoS attack

Starting DDoS attack using LOIC tool is easy because of the provided simple GUI. However, the tool is not flexible to generate DDoS traffic in a specific magnitude and it exhausts the processing resource of the system. As a result, it was difficult to do other tasks while it is running. For this reason we have used hping3 to generate both UDP and TCP-SYN flood attacks. The commands used to start both attacks are:

```
hping3 --udp --data 32 --flood 10.10.10.172 -p
1234 --rand-source
hping3  -d 120 -S -w 64 -p 2323 --flood --rand-source
10.10.10.172
```

The first line of command is used to send UDP flood to victim network with parameters: data size 32 + header 28 total of 60 bytes, destination port 1234, and with random IP sources. The magnitude of the traffic is about 180 kpps for one instance of the above command and multiple instances are used to increase the attack magnitude. The second line is used to start TCP-SYN attack with parameters: TCP-SYN flag, a window size of 64, a packet size of 120 bytes. A single instance can generate about 152 kpps in our testbed.

5.4.3 Generating normal traffic

Normal traffic is generated using the iperf3 tool. The server-side script of iperf3 is used to run on victim system and the client side script is on the normal traffic generator system. The command used to generate and receive are:

```
iperf3 -s
iperf3 -c 10.10.10.172 -p 5201 -t 60
```

A single instance of the above iperf3 command can generate about 28 kpps. Pkt-gen tool is used to test the maximum throughput of fastnetmon in a given hardware configuration. The command used to generate and receive are:

```
    pkt-gen -f tx -l 60 #Send packets in high speed

    pkt-gen -f rx       #receive and display the magnitude of
                        #the  the incoming traffic
```

5.4.4 Scrips developed for measurements and validation

We have developed scripts to measure and cross-validate the traffic size and magnitude generated and received. The bash script is added to Appendix A displays the current network usage in bytes per second and packets per second.

Chapter 6
Results

6.1 Traffic input

The results discussed in this chapter are based on the traffic input parameters summarized in Table 6.1. We have used tools and scripts presented in previous chapters to generate traffic and measure the experiment outputs.

Table 6.1: The traffic inputs used in our experiment .

Scenarios	kpps	packet size	mbps
Normal Traffic	28	1448	324.5
UDP flood attack case 1	180	60	86.4
UDP flood attack case 2	360	60	172.8
TCP-SYN attack	152	120	145.92

6.2 Maximum throughput

Using hardware configuration in Table 5.1. First, we have measured the maximum throughput capability of a given hardware and operating system before starting the detection application. This is is used as a benchmark to generate packets for other test scenarios and to understand the limitation of our testbed environment in terms of link speed, bus or hardware limitations. The maximum throughput of the detection hardware tested in interface bridge-mode with small raw packet size using pkt-gen application is 850218 pps . Table 6.2 shows summary of different packet sizes throughput.

The result in Figure 6.1 shows the throughput in pps and FastNetMon CPU usage in percentage for the 4 types of packet sizes. Especially, in the case of a traffic with packet size 60 bytes the CPU percentage reached to 100 %. In other test cases, with byte sizes from 500 - 1500 we reached to the hardware limitation before CPU peak.

Table 6.2: Maximum throughput result using interface in bridge mode .

No.	packet size	output kpps	mbps
1	60	850.1	408
2	500	191.2	760
3	1000	100.1	803
4	1500	70	842

The memory usage was constant for all types of scenarios because of fastnetmon memory usage, as described in Section 4.2.2.

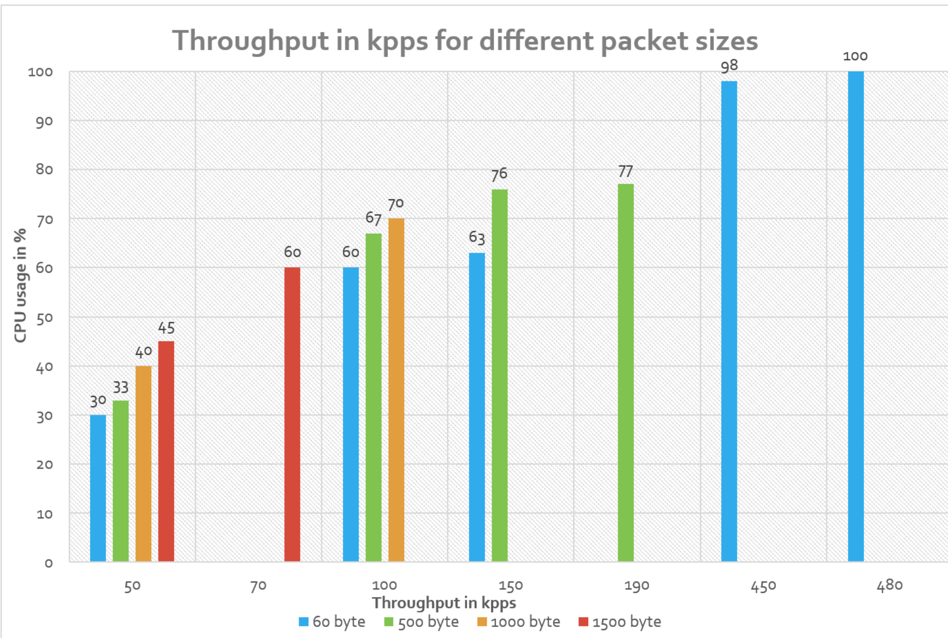


Figure 6.1: Fastnetmon detection throughput for different packet size and pps

Another throughput test case is measured using fastnetmon on different hardware with better processing capacity, which is presented in Figure 4.7. The test case is intended to study the throughput of fastnetmon in different CPU types. The result is shown in Table 6.3.

Table 6.3: Fastnetmon Maximum throughput results in different processors.

CPU type	packet size	output kpps	mbps
Intel Core i7 CPU 860 @2.80Ghz	60	~480.1	230.4
Intel Xeon(R) CPU @3.40Ghz	60	~790.2	379.2

6.3 Packet Delay

The packet delay generally occurs weather in copying packets from NIC to user-space or processing packet in detection application, as described in Section 2.6. We have developed a script to measure the time detection application takes to process each packets, but the result couldn't give us reasonable output so that we didn't include in this section. We have used ping utility to test packet delay. There reference Round-Trip Time (RTT) is measured before starting the fastnetmon and generating packets. Afterwards, packet delay is measured for the defined scenarios while fastnetmon is running. The reference packet delay is displayed in Table 6.4. Every packet delay measurement is a average of a round trip time in Milli seconds (ms) after running the ping utility for 60 seconds between the normal traffic generator system and victim system.

Table 6.4: Reference round trip time in ms between victim and normal traffic generator system.

	Minimum	Average	Maximum	standard deviation
RTT in ms	1.51	4.46	6.46	1.3

The packet delay under normal operation and the reference are close to each other. The offered traffic load as normal traffic in a given hardware didn't affect the latency of the packet as shown in Figure 6.3. For the UDP flood attack case 1 packet delay remain almost the same as the reference and normal operation cases.

On the rest of the two scenarios, under UDP flood case 2 and TCP-SYN flooding the detection engine experienced packet delay causing packet drops. The UDP flood attack mixed with normal traffic let the CPU to reach the peak and started dropping packets. The TCP-SYN flood cause packet delay and loss because the detection engine requires more time to check TCP packet than UDP.

The packet loss caused by the packet delay and CPU overload by DDoS attacks are summarized in the Table 6.5. The packet loss measurements are done for the normal traffic which is used analyze the legitimate user requests lost.

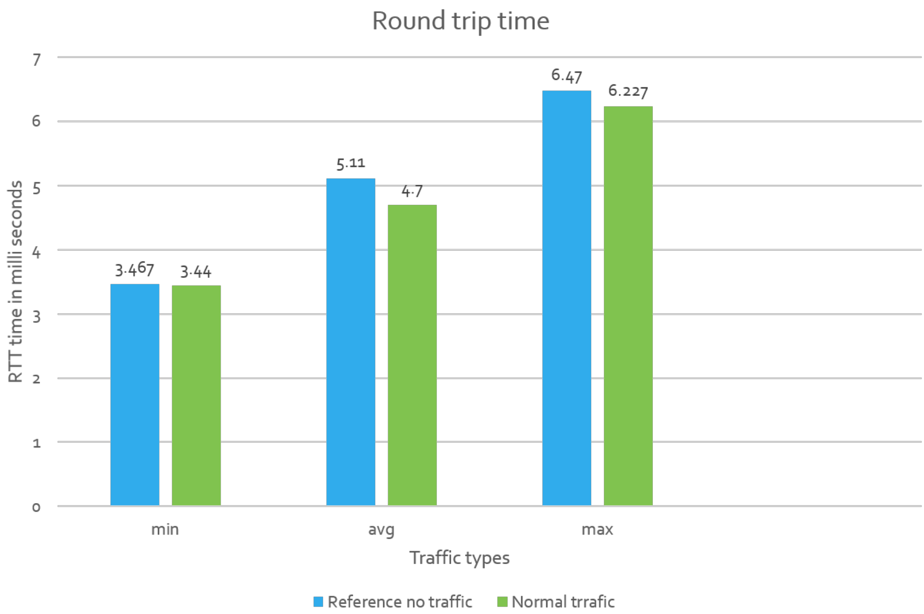


Figure 6.2: Packet delay under normal traffic versus the reference RTT

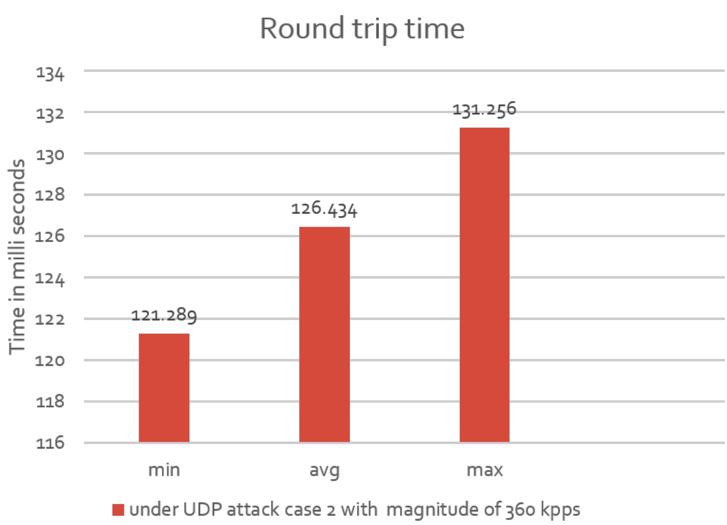


Figure 6.3: Packet delay under UDP flood attack case 2

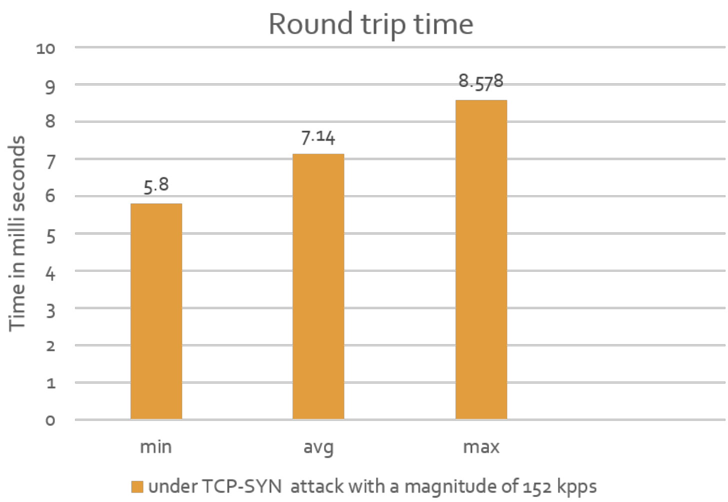


Figure 6.4: Packet delay under TCP-SYN flood

Table 6.5: packet loss during TCP-SYN and UDP flood attack .

ATTACK TYPE	Normal Traffic sent	sent packets	dropped packets	lost in %
UDP flood	UDP	1720455	771441	45%
TCP-SYN FLOOD	UDP	1627694	427700	26%

6.4 Resource usage

The CPU and RAM utilization is measured for the three types of scenarios. The results of measurements for the scenarios are summarized inTable 6.6.

Table 6.6: Resource utilization and traffic load of the scenarios.

Scenario	CPU %	RAM %	kpps
Under Normal	10	0.1	28
UDP flood case 1	74	0.1	180
UDP flood case 2	100	0.1	360
TCP-SYN flood	70	0.1	152

Another CPU utilization measurement is done using different packets size in the the same kpps magnitude. The details of traffic magnitude and the obtained results for all types of packet sizes could be found in appendix. The only result we could obtained before reaching to the reference benchmark of the hardware limitation is for the traffic magnitude of 50 kpps, as displayed in Figure 6.5.

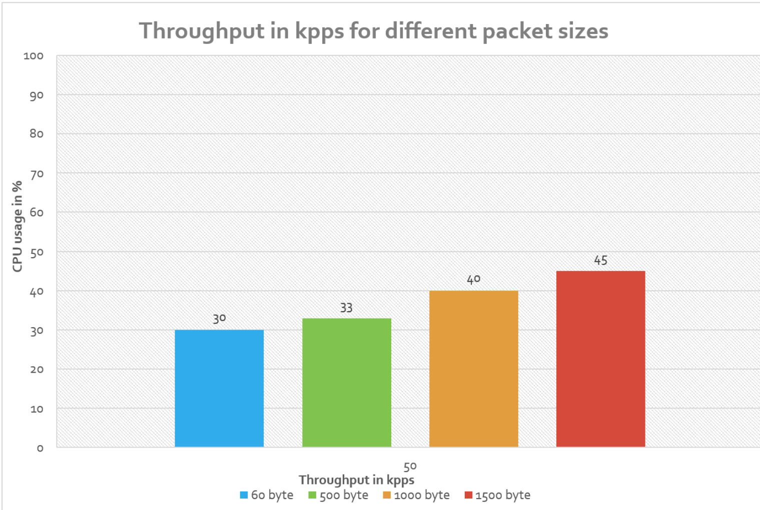


Figure 6.5: CPU usage and throughput for different packet sizes under 50 kpps

6.5 Detection accuracy

Detection accuracy results are measured using tools: mainly tshark, nload, iperf and our developed script. In our test environment we have measured traffic in each systems in terms of packet number , magnitude and direction for all the scenarios. Direction is a traffic direction to identify its source and destination. Tshark can filter and dump packets based on their Physical (MAC) and IP address, so that we could easily identify the traffic amount and it's direction.

The traffic is filtered based on the report of fastnetmon, which is detected anomaly and signature based against a given normal network traffic profile. The result displayed in Table 6.7 shows the received traffic measured in victim system for each scenarios.

Table 6.7: Detection accuracy output.

scenarios	Generated traffic	source	Received in Victim system %
under Normal	1720455 packets	normal traffic generator	1720455 packets
under UDP flood case 1	180 kpps of size 60	DDoS attack generator	0
under UDP flood case 2	360 kpps of size 60	DDoS attack generator	0
under TCP-SYN flood	152 kpps of size 120	DDoS attack generator	0

Table 6.7 shows filtering based on fastenetmon report and based on the accuracy equation (5.1).

Chapter 7

Discussion

The results obtained from the performance and detection accuracy measurements are presented in the previous chapter. This chapter is a discussion of the obtained results.

7.1 Maximum throughput

The first throughput test aim was to analyze the maximum packet capturing and forwarding capability of a given hardware and the packet capturing framework used. This result is compared with the fastnetmon throughput in the same testing environment using different traffic magnitude. Fastnetmon CPU utilization reached to 100 % for all types of processors used while processing the traffic magnitude of 50 kpps.

As shown in Figure 7.1 Fastnetmon throughput for different CPU is different. This indicates fastetmon throughput performance is directly related to the processor speed of a given hardware. Maximum throughput was also studied by saboor [SAA13] as maximum packet rate in two different CPU cores and it has been found that snort packet handling capacity is doubled. Their result shows that in a processor with 2.04 Ghz and 3.40 ghz are 950 pps and 1700 pps, respectively. This also indicates that fastnetmon with improved PCE handles significantly high traffic rate than snort.

7.2 Packet delay

Packet delay is measured to analyze the effect of DDoS traffic on fastnetmon. The reference RTT is measured before starting packet generator and fastnetmon to compare with the scenarios packet delay. Under normal operation and UDP flood case1 the packet delay is not affected by the load of the traffic. Under UDP flood case 2 the CPU reached its peak performance. As a result, packets are started to

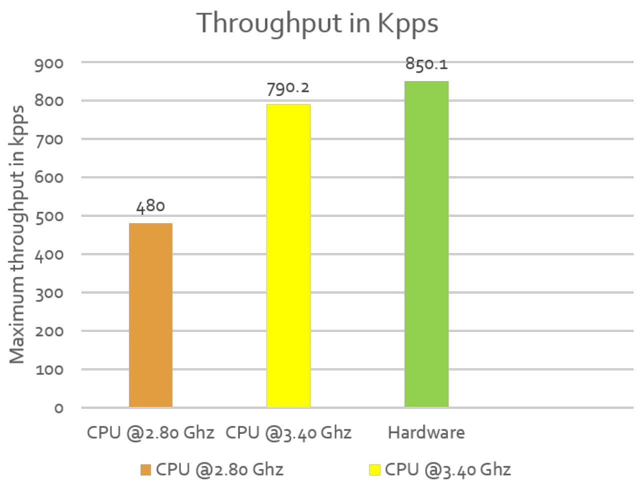


Figure 7.1: Comparing the hardware throughput vs fastnetmon in different processors

drop before being processed and forwarded. This indicates how UDP flood can easily overwhelms network level forwarding devices and resources.

Under TCP-SYN flood the result shows that fastnetmon begins to drop packets while the CPU usage was at 70 % as shown in Figure 6.4. The cause of packet delay in this scenario is the fact that fastnetmon takes more time to process TCP packets than other protocols like UDP. While most of the packets in fastnetmon including UDP are checked anomaly using packet size and rate, for TCP packets additional header values like the TCP flags are checked to detect TCP-SYN flood. As a result the incoming packets are dropped because the packet processing speed in a given hardware couldn't cope up with the incoming traffic speed.

7.3 Resource usage

Table 6.6 shows the resource utilization of fastnetmon for the scenarios. The memory utilization of fastnetmon is constant through out the experiment. Fastnetmon allocates memory during the initial state for the hosts in victim network.

We have observed that CPU utilization is vary from scenarios to scenarios and for traffic in different packet sizes. Under normal operation our test was based on client to server data exchange over UDP protocol. The average transaction magnitude was about 28 kpps using UDP data size 1448 and the average CPU usage was 10 % of the the given hardware configuration.

Under UDP flood case 2 the victim network couldn't response to almost 50 % of the requests from the normal traffic generator, as displayed Table 6.5. The CPU utilization of fastnetmon hardware increased to 100 % as soon son as the UDP flood is reached and it stayed same for the whole test duration. Under TCP-SYN flood the CPU utilization of fastnetmon was 70%.

Another important observation during the experiment is that the CPU utilization for different packet sizes under the same traffic magnitude results in different CPU utilization. Figure 6.5 shows, when the packet size increases the CPU utilization increases slightly. It indicates that even if fastnetmon process the same types of packet in a uniform format, CPU utilization is affected by the size of packet because packet length is the size of buffer on physical memory which CPU handles.

7.4 Detection Accuracy

To measure detection accuracy we have developed a script that makes traffic filter policy from fastnetmon attack report file. The modified Fastnetmon uses to filter the incoming traffic based on a given filter policy. The accuracy result for all scenarios was 100 % accurate, as shown in Table 5.1.

Accuracy result shows that the possibility of DDoS attack mitigation using fastnetmon if it is configured properly. We have implemented simple filtering algorithm based on pcap packet filtering API. While this is enough to test the accuracy of the fastnetmon detection, it is not standard and best practice of packet filtering.

Another important thing we have observed during traffic filtering is that filtering traffic before it enters to the victim network prevents resources from being overwhelmed by DDoS flooding attacks. This was witnessed when we were able to filter during both TCP and UDP attack scenarios. We have used pre-filtering techniques of libpcap PCE, which can filter packets before it passes to traffic processing application, fastnetmon detection engine. As a result,the CPU utilization of fastnetmon remain same as normal operation.

7.5 Summary

Overall, we are satisfied with our experimental testbed setup and the obtained results. However, we are aware of that the result could be better if the test environment is not limited by hardware and software constraints and also it has some imperfections in the implementation. The result and the gained experience could be used as the basis for the conclusion for this thesis. We hope improving the testbed developed in thesis can be used as a benchmark for future studies.

Chapter 8
Conclusion

In this thesis, we have studied the possibility of software-based DDoS detection in a commodity hardware. There has been much research in this topic and we have taken advantage of several previously proposed solutions and suggestion in the areas of traffic capturing, measuring and detection of denial of service attacks. We have developed an experimental testbed consists of a chosen DDoS detection, DDoS attack tools, and commodity hardware and carried out performance and detection accuracy evaluation.

In chapter 4, the chosen DDoS detection application, fastnetmon, is presented. This includes the PCE to capture traffic, detection methods and capability, and deployment architecture it supports, as well as the modification we made to the fastnetmon software architecture. An introduction to traffic generator and DDoS attack tools were also provided to give the reader an understanding of the functionality, capability, and limitation of the tools. Finally, the complete experimental testbed setup is presented.

Evaluation methodology for the testbed in terms of performance and detection accuracy is presented in chapter 5. The well-defined metrics and scenarios are described including the hardware configuration used in the testbed.

The experimental result has shown that open source based DDoS detection performance can be increased in commodity hardware by utilizing the available hardware resources. This was witnessed by using netmap, one of the packet capturing improvements done for modern commodity hardware to achieve fast packet capturing. We have shown that, an inexpensive commodity hardware and fastnetmon that we have used in our testbed can process and forward small-sized packets at several kpps using only a single core processor. A similar research on performance analysis of snort against DDoS in a similar hardware without improved PCE carried out by Saboor [SAA13]. The results we obtained is much higher than their result. Therefore, we recommend using improved PCE frameworks to achieve a better performance in traffic processing capacity in a commodity hardware.

While processing network traffic inline, packet delay can be caused by PCE or processing application. We have observed that processing TCP packets than UDP and larger packets than smaller has a direct impact on packet delay. As a result, the processing delay increased the RTT and decrease the throughput performance of the fastnetmon in a given hardware configuration and also cause packet loss. The same is true for detection methods used, signature-based detection takes more time to process packet than anomaly detection. It indicates that relying more on anomaly detection than signature-based detection can also improve the throughput performance in inline deployment.

The experiment practically showed that the fastetmon detection performance is directly related to the CPU processing capacity of a given hardware. Comparing the result obtained from the experiment while using two different processor, it shows that changing the CPU speed of 2.40 GHz processor to 3.80 GHz processor the throughput performance increased by 40%. The RAM usage remained constant. Therefore, the performance of DDoS detection can be improved by using CPU with a higher processing capacity and multi-core.

The detection accuracy of fastnetmon observed in the experiment shows that fastnetmon is efficient. Fastnetmon anomaly detection method detects DDoS by comparing previously defined patterns of normal traffic rate in terms of pps, mbps and number of flows with anomalies of network traffic. According to the research work by Cviti'c et al. [CPPM] detection accuracy of using such approach to detect DDoS is about 98 % accurate. This indicates that fastnetmon is using an effective way of detecting DDoS attack.

Based on findings during the work with this thesis, we conclude that commodity hardware with effective DDoS detection application like fastnetmon and improved fast packet capturing frameworks such as netmap and PF_Ring ZC, has a potential to be used as DDoS defense mechanism in victim end.

8.1 Future Work

Due to the available hardware and software limitation during this thesis work, we have used a single core of the hardware. It may also be helpful to use better software architecture and higher giga interface links in order to take full advantage of modern multi core processing power of commodity hardware.

we propose a suggestion for further work:

- **Increase the number of CPU core used:** Modern commodity hardware have multple multi-core CPUs. It would be interesting to see the performance of DDoS detection by leveraging the CPU power provided by such hardware.

References

[AMAM+11] Hashem Alaidaros, Massudi Mahmuddin, Ali Al-Mazari, et al. An overview of flow-based and packet-based intrusion detection performance in high speed networks. 2011.

[AR12] Mohammed Alenezi and M Reed. Methodologies for detecting dos/ddos attacks against network servers. In *Proceedings of the Seventh International Conference on Systems and Networks Communications—ICSNC*, 2012.

[BDKC10] Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, and Georg Carle. Comparing and improving current packet capturing solutions based on commodity hardware. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 206–217. ACM, 2010.

[BSA+16] Akashdeep Bhardwaj, GVB Subrahmanyam, Vinay Avasthi, Hanumat Sastry, and Sam Goundar. Ddos attacks, new ddos taxonomy and mitigation solutions—a survey. In *Signal Processing, Communication, Power and Embedded System (SCOPES), 2016 International Conference on*, pages 793–798. IEEE, 2016.

[CDL16] Alfredo Cardigliano, Luca Deri, and Tord Lundstrom. Commoditising ddos mitigation. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2016 International*, pages 523–528. IEEE, 2016.

[CPPM] Ivan Cvitić, Dragan Peraković, Marko Periša, and Mario Musa. Network parameters applicable in detection of infrastructure level ddos attacks. *network*, 82:1.

[D+04] Luca Deri et al. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93. Amsterdam, Netherlands, 2004.

[Der03] Luca Deri. Passively monitoring networks at gigabit speeds using commodity hardware and open source software. In *Proceedings of the Passive and Active Measurement Conference*, pages 1–7, 2003.

[DHKB16] Ashaq Hussain Dar, Beenish Habib, Farida Khurshid, and M Tariq Banday. Experimental analysis of ddos attack and it's detection in eucalyptus private cloud platform. In *Advances in Computing, Communications and Informatics (ICACCI), 2016 International Conference on*, pages 1718–1724. IEEE, 2016.

[DM04] Christos Douligeris and Aikaterini Mitrokotsa. Ddos attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643–666, 2004.

[fas] fastnetmon. Bgp flow spec. https://fastnetmon.com/docs/bgp_flow_spec/. Accessed: 2-12-2017.

[For] Internet Engineering Task Force. Data center benchmarking terminology. https://tools.ietf.org/html/rfc8238#page-10. Accessed: 31-01-2018.

[HAG10] SHC Haris, RB Ahmad, and MAHA Ghani. Detecting tcp syn flood attack based on anomaly detection. In *Network Applications Protocols and Services (NETAPPS), 2010 Second International Conference on*, pages 240–244. IEEE, 2010.

[Hpi] Hping. active network security tool. http://www.hping.org. Accessed: 12-12-2017.

[HSS12] Mohd Jameel Hashmi, Manish Saxena, and Rajesh Saini. Classification of ddos attacks and their defense techniques using intrusion prevention system. *International Journal of Computer Science and Communication Networks*, 2(5):607–14, 2012.

[hto] htop. htop. https://hisham.hm/htop/. Accessed: 2-11-2017.

[Imp] Imperva. Tcp syn flood. https://www.incapsula.com/ddos/attack-glossary/syn-flood.html. Accessed: 5-12-2017.

[Inc] Cloudflare Inc. Udp flood attack. https://www.cloudflare.com/learning/ddos/udp-flood-ddos-attack/. Accessed: 5-12-2017.

[ipe] iperf3. https://iperf.fr/. Accessed: 2016-11-06.

[JHP+16] Grafika Jati, Budi Hartadi, Akmal Gafar Putra, Fahri Nurul, M Riza Iqbal, and Setiadi Yazid. Design ddos attack detector using ntopng. In *Big Data and Information Security (IWBIS), International Workshop on*, pages 139–144. IEEE, 2016.

[KB12] Hirak Jyoti Kashyap and DK Bhattacharyya. A ddos attack detection mechanism based on protocol specific traffic features. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, pages 194–200. ACM, 2012.

[KS11] P Arun Raj Kumar and S Selvakumar. Distributed denial of service attack detection using an ensemble of neural classifier. *Computer Communications*, 34(11):1328–1341, 2011.

[Lov] Robert M. Love. Linux man page. https://linux.die.net/man/1/taskset. Accessed: 2018-02-06.

[LSBM15] Jessica A Lopez, Yali Sun, Peter B Blair, and M Shahid Mukhtar. Tcp three-way handshake: linking developmental processes with plant immunity. *Trends in plant science*, 20(4):238–245, 2015.

[LTDa] Fastnetmon LTD. Fastnetmon. https://fastnetmon.com/screenshoots/. Accessed: 17-1-2018.

[LTDb] Fastnetmon LTD. Fastnetmon. https://fastnetmon.com. Accessed: 19-12-2017.

[Mer17] Mehdi Merouane. An approach for detecting and preventing ddos attacks in campus. *Automatic Control and Computer Sciences*, 51(1):13–23, 2017.

[MPR03] Jelena Mirkovic, Gregory Prier, and Peter Reiher. Source-end ddos defense. In *Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on*, pages 171–178. IEEE, 2003.

[MR04] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[Net] Arbor Networks. Ddos attacks and rise of iot botnets. https://ripe75.ripe.net/presentations/53-RIPE75-DDoS-and-Rise-of-IOT-botnets.pdf. Accessed: 2-12-2017.

[NIG+17] Kseniya Yu Nikolskaya, Sergey A Ivanov, Valentin A Golodov, Aleksey V Minbaleev, and Gregory D Asyaev. Review of modern ddos-attacks, methods and means of counteraction. In *" Quality Management, Transport and Information Security, Information Technologies"(IT&QM&IS), 2017 International Conference*, pages 87–89. IEEE, 2017.

[nLa] ntop LTD. nscrub. https://www.ntop.org/products/ddos-mitigation/nscrub/. Accessed: 31-01-2018.

[nLb] ntop LTD. ntop. https://www.ntop.org/products/traffic-analysis/ntop/. Accessed: 31-01-2018.

[nLc] ntop LTD. Pf ring. https://www.ntop.org/products/packet-capture/pf_ring/. Accessed: 31-01-2018.

[NSCP15] Bharti Nagpal, Pratima Sharma, Naresh Chauhan, and Angel Panesar. Ddos tools: Classification, analysis and comparison. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 342–346. IEEE, 2015.

[Nto] Ntop. ndpi. https://www.ntop.org/products/deep-packet-inspection/ndpi/. Accessed: 12-12-2017.

[PLR07] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Computing Surveys (CSUR)*, 39(1):3, 2007.

[PZC+96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, 1996.

[RDC12] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 gbit/s line rate packet processing using commodity hardware: Survey and new proposals, 2012.

[RGMFGT13] Rafael A Rodríguez-Gómez, Gabriel Maciá-Fernández, and Pedro García-Teodoro. Survey and taxonomy of botnet research through life-cycle. *ACM Computing Surveys (CSUR)*, 45(4):45, 2013.

[Rie] Roland Riegel. Linux man page. https://linux.die.net/man/1/nload. Accessed: 2-11-2017.

[Riz12] Luigi Rizzo. Netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[SAA13] Amtul Saboor, Monis Akhlaq, and Baber Aslam. Experimental evaluation of snort against ddos attacks under different hardware configurations. In *Information Assurance (NCIA), 2013 2nd National Conference on*, pages 31–37. IEEE, 2013.

[Sea] Tara Seals. Ddos-for-hire costs just $38 per hour. https://www.infosecurity-magazine.com/news/ddosforhire-costs-just-38-per-hour/. Accessed: 9-12-2017.

[Sno] Snort. What is snort. https://www.snort.org/#documents. Accessed: 05-01-2018.

[tB17] Bram ter Borch. Session based high bandwidth throughput testing. 2017.

[Wir] Wireshark. tshark. https://www.wireshark.org/docs/man-pages/tshark.html. Accessed: 2-12-2017.

[XMZ09] Rui Xu, Wen-li Ma, and Wen-ling Zheng. Defending against udp flooding by negative selection algorithm based on eigenvalue sets. In *Information Assurance and Security, 2009. IAS'09. Fifth International Conference on*, volume 2, pages 342–345. IEEE, 2009.

[ZJT13] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

Appendix A

Script for traffic monitoring

```
INTERVAL="1"  # update interval in seconds
if [ -z "$1" ]; then
  echo
  echo usage: $0 [network-interface]
  echo
  echo e.g. $0 eth0
  echo
  echo shows packets-per-second
  exit
fi

IF=$1
while true
do
  RPP1=`cat /sys/class/net/$1/statistics/rx_packets`
  TPP1=`cat /sys/class/net/$1/statistics/tx_packets`

  RBP1=`cat /sys/class/net/$1/statistics/rx_bytes`
  TBP1=`cat /sys/class/net/$1/statistics/tx_bytes`
  sleep $INTERVAL

  RPP2=`cat /sys/class/net/$1/statistics/rx_packets`
  TPP2=`cat /sys/class/net/$1/statistics/tx_packets`

  RBP2=`cat /sys/class/net/$1/statistics/rx_bytes`
  TBP2=`cat /sys/class/net/$1/statistics/tx_bytes`
  TXPPS=`expr $TPP2 - $TPP1`
  RXPPS=`expr $RPP2 - $RPP1`

  TXBPS=`expr $TBP2 - $TBP1`
  TXBPS=`expr $TXBPS \* 8`

  RXBPS=`expr $RBP2 - $RBP1`
  RXBPS=`expr $RXBPS \* 8`


  echo "TX PPS $1: $TXPPS pkts/s RX PPS $1: $RXPPS pkts/s"
```

```
  echo "TX BPS $1: $TXBPS bits/s RX BPS $1: $RXBPS bits/s"
done
```

Listing A.1: traffic monitoring bash script

```
\label{appendix:filterreport}
grep -rnw '/var/log/fastnetmon_attacks/' -e '> 10.10.10' | awk '{print
     $5}' | sort | uniq > /root/ddos_measrments/folter_list.txt
sed -i  's/:/ and not dst port /'  /root/ddos_measrments/folter_list.
    txt
sed -i  's/129.241/ ( not dst 129.241/'  /root/ddos_measrments/
    folter_list.txt
sed -e ':a' -e 'N' -e '$!ba' -e 's/\n/ ) or  /g '  /root/
    ddos_measrments/folter_list.txt > /root/ddos_measrments/
    final_list2.txt
echo ')'>> /root/ddos_measrments/final_list2.txt
sed -e ':a' -e 'N' -e '$!ba' -e 's/\n/  /g '  /root/ddos_measrments/
    final_list2.txt > /root/ddos_measrments/filter.txt
sed -e ':a' -e 'N' -e '$!ba' -e 's/\n/  /g '  /root/ddos_measrments/
    filter.txt > /root/ddos_measrments/filter2.txt
/root/ddos_measrments/send_reciev/compile.sh
```

Listing A.2: script developed for retrieving filter policy from fastnetmon report

Appendix B

Modified source code for fastnetmon

```

/*****************************************************************
* file:   netmap_plugin.c
* date:   2018-Feb-14 12:14:19 AM
* Author: Meklit Elfiyos
* Last Modified:2018-Feb-14 12:14:19 AM
*
* Description: fastnetmon pcap based \gls{pce}
*****************************************************************/
#include <sys/types.h>
#include <pcap.h>
#include <netinet/if_ether.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <stdio.h>
#include <sys/time.h>
// log4cpp logging facility
#include "log4cpp/Category.hh"
#include "log4cpp/Appender.hh"
#include "log4cpp/FileAppender.hh"
#include "log4cpp/OstreamAppender.hh"
#include "log4cpp/Layout.hh"
#include "log4cpp/BasicLayout.hh"
#include "log4cpp/PatternLayout.hh"
#include "log4cpp/Priority.hh"

#include <boost/version.hpp>
#include <boost/algorithm/string.hpp>

#include "../fast_library.h"

// For support uint32_t, uint16_t
#include <sys/types.h>

// For config map operations
```

```
#include <string>
#include <map>

#include <stdio.h>
#include <iostream>
#include <string>
#define NETMAP_WITH_LIBS

// Disable debug messages from Netmap
#define NETMAP_NO_DEBUG
#include <net/netmap_user.h>
#include <boost/thread.hpp>

#if defined(__FreeBSD__)
// On FreeBSD function pthread_attr_setaffinity_np declared here
#include <pthread_np.h>

// Also we have different type name for cpu set's store
typedef cpuset_t cpu_set_t;
#endif

#include "../fastnetmon_packet_parser.h"

#include "../unified_parser.hpp"

// For pooling operations
#include <poll.h>

// For support: IPPROTO_TCP, IPPROTO_ICMP, IPPROTO_UDP
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "netmap_collector.h"
//====================================
pcap_t* descr_snd=NULL;
//====================================

// By default we read packet size from link layer
// But in case of Juniper we could crop first X bytes from packet:
// maximum-packet-length 110;
// And this option become mandatory if we want correct bps speed in toolkit
bool netmap_read_packet_length_from_ip_header = false;

uint32_t netmap_sampling_ratio = 1;
unsigned long max=0,min=0,df=0;
/* prototypes */
void netmap_thread(struct nm_desc* netmap_descriptor, int netmap_thread);
void consume_pkt(u_char* buffer, int len, int thread_number);

```

```
// Get log4cpp logger from main program
extern log4cpp::Category& logger;

// Pass unparsed packets number to main program
extern uint64_t total_unparsed_packets;

// Global configuration map
extern std::map<std::string, std::string> configuration_map;

u_int num_cpus = 0;

// This variable name should be uniq for every plugin!
process_packet_pointer netmap_process_func_ptr = NULL;

bool execute_strict_cpu_affinity = true;

int receive_packets(struct netmap_ring* ring, int thread_number) {
    u_int cur, rx, n;

    cur = ring->cur;
    n = nm_ring_space(ring);

    for (rx = 0; rx < n; rx++) {
        struct netmap_slot* slot = &ring->slot[cur];
        char* p = NETMAP_BUF(ring, slot->buf_idx);

        // process data
        consume_pkt((u_char*)p, slot->len, thread_number);

        cur = nm_ring_next(ring, cur);
    }

    ring->head = ring->cur = cur;
    return (rx);
}

void consume_pkt(u_char* buffer, int len, int thread_number) {
    // We should fill this structure for passing to FastNetMon
    simple_packet packet;
  packet.sample_ratio = netmap_sampling_ratio;

    if (!parse_raw_packet_to_simple_packet(buffer, len, packet,
    netmap_read_packet_length_from_ip_header)) {
        total_unparsed_packets++;

        return;
    }
//========================= Packet Delay calculator ==========
  struct timespec tps, tpe;
  if (clock_gettime(CLOCK_REALTIME, &tps) != 0)
  {
    perror("clock_gettime");
```

```

  }
  netmap_process_func_ptr(packet);

  if (clock_gettime(CLOCK_REALTIME, &tpe) != 0)
  {
    perror("clock_gettime");

  }

  else
    if((df=tpe.tv_nsec-tps.tv_nsec)> max)
      max=df;

    printf("%lu s, %lu ns\n", tpe.tv_sec-tps.tv_sec,);

  gettimeofday(&after , NULL);
  before_detection_ms = (double)before.tv_sec + (double)before.tv_usec;
  after_detection_ms = (double)after.tv_sec + (double)after.tv_usec;
  diff = (double)after_detection_ms - (double)before_detection_ms;
  printf("Total time elapsed : %.0lf us\n" , diff );
//=====================================================================
//=====================================================================
if(descr_snd== NULL)
  {

    char errbf[PCAP_ERRBUF_SIZE];
    char filter_exp[255] = "";
    struct bpf_program filter;
    bpf_u_int32 subnet_mask, ip;
    descr_snd = pcap_open_live("enp0s25",1500,1,-1,errbf);
    if(descr_snd == NULL)
    {
      printf("pcap_open_live(): At sending");
    //return;
    }
    if (pcap_compile(descr_snd, &filter, filter_exp, 0, ip) == -1)
    {
      printf("Bad filter - %s\n", pcap_geterr(descr_snd));

    }
    if (pcap_setfilter(descr_snd, &filter) == -1)
    {
      printf("Error setting filter - %s\n", pcap_geterr(descr_snd));
    }


  }
  pcap_inject(descr_snd, &p, slot->len);
//====================================================================
}
```

```

void receiver(std::string interface_for_listening) {
    struct nm_desc* netmap_descriptor;

    struct nmreq base_nmd;
    bzero(&base_nmd, sizeof(base_nmd));

    // Magic from pkt-gen.c
    base_nmd.nr_tx_rings = base_nmd.nr_rx_rings = 0;
    base_nmd.nr_tx_slots = base_nmd.nr_rx_slots = 0;

    std::string interface = "";
    std::string system_interface_name = "";
    // If we haven't netmap: prefix in interface name we will append it
    if (interface_for_listening.find("netmap:") == std::string::npos) {
        system_interface_name = interface_for_listening;

        interface = "netmap:" + interface_for_listening;
    } else {
        // We should skip netmap prefix
        system_interface_name = boost::replace_all_copy(interface_for_listening, "netmap:", "");

        interface = interface_for_listening;
    }

#ifdef __linux__
    manage_interface_promisc_mode(system_interface_name, true);
    logger.warn("Please disable all types of offload for this NIC manually: ethtool -K %s gro off gso off tso off lro off", system_interface_name.c_str());
#endif

    netmap_descriptor = nm_open(interface.c_str(), &base_nmd, 0, NULL);

    if (netmap_descriptor == NULL) {
        logger.error("Can't open netmap device %s", interface.c_str());
        exit(1);
        return;
    }

    logger.info("Mapped %dKB memory at %p", netmap_descriptor->req.nr_memsize >> 10, netmap_descriptor->mem);
    logger.info("We have %d tx and %d rx rings", netmap_descriptor->req.nr_tx_rings,
                netmap_descriptor->req.nr_rx_rings);

    if (num_cpus > netmap_descriptor->req.nr_rx_rings) {
```

```
        num_cpus = netmap_descriptor->req.nr_rx_rings;

        logger.info("We have number of CPUs bigger than number of NIC RX queues. Set number of "
                    "CPU's to number of threads");
    }

    /*
        protocol stack and may cause a reset of the card,
        which in turn may take some time for the PHY to
        reconfigure. We do the open here to have time to reset.
    */

    int wait_link = 2;
    logger.info("Wait %d seconds for NIC reset", wait_link);
    sleep(wait_link);

    boost::thread_group packet_receiver_thread_group;

    for (int i = 0; i < num_cpus; i++) {
        struct nm_desc nmd = *netmap_descriptor;
        // This operation is VERY important!
        nmd.self = &nmd;

        uint64_t nmd_flags = 0;

        if (nmd.req.nr_flags != NR_REG_ALL_NIC) {
            logger.error("Ooops, main descriptor should be with NR_REG_ALL_NIC flag");
        }

        nmd.req.nr_flags = NR_REG_ONE_NIC;
        nmd.req.nr_ringid = i;

        /* Only touch one of the rings (rx is already ok) */
        nmd_flags |= NETMAP_NO_TX_POLL;

        struct nm_desc* new_nmd =
        nm_open(interface.c_str(), NULL, nmd_flags | NM_OPEN_IFNAME | NM_OPEN_NO_MMAP, &nmd);

        if (new_nmd == NULL) {
            logger.error("Can't open netmap descriptor for netmap per hardware queue thread");
            exit(1);
        }

        logger.info("My first ring is %d and last ring id is %d I'm thread %d",
                    new_nmd->first_rx_ring, new_nmd->last_rx_ring, i);


```

```
        /*
        logger<< log4cpp::Priority::INFO<< "We are using Boost "
            << BOOST_VERSION / 100000     << "."  // major version
            << BOOST_VERSION / 100 % 1000 << "."  // minior version
            << BOOST_VERSION % 100;
        */

        logger.info("Start new netmap thread %d", i);

// Well, we have thread attributes from Boost 1.50

#if defined(BOOST_THREAD_PLATFORM_PTHREAD) && BOOST_VERSION / 100 % 1000 >= 50 && !defined(__APPLE__)
        /* Bind to certain core */
        boost::thread::attributes thread_attrs;

        if (execute_strict_cpu_affinity) {
            cpu_set_t current_cpu_set;

            int cpu_to_bind = i % num_cpus;

            CPU_ZERO(&current_cpu_set);
            // We count cpus from zero
            CPU_SET(cpu_to_bind, &current_cpu_set);

            logger.info("I will bind this thread to logical CPU: %d", cpu_to_bind);

            int set_affinity_result = pthread_attr_setaffinity_np(thread_attrs.native_handle(), sizeof(cpu_set_t), &current_cpu_set);

            if (set_affinity_result != 0) {
                logger.error("Can't specify CPU affinity for netmap thread");
            }
        }

        // Start thread and pass netmap descriptor to it
        packet_receiver_thread_group.add_thread(
        new boost::thread(thread_attrs, boost::bind(netmap_thread, new_nmd, i)));
#else
        logger.error("Sorry but CPU affinity did not supported for your platform");
        packet_receiver_thread_group.add_thread(new boost::thread(netmap_thread, new_nmd, i));
#endif
    }

    // Wait all threads for completion
    packet_receiver_thread_group.join_all();
```

```
}

void netmap_thread(struct nm_desc* netmap_descriptor, int thread_number) {
    struct nm_pkthdr h;
    u_char* buf;
    struct pollfd fds;
    fds.fd = netmap_descriptor->fd; // NETMAP_FD(netmap_descriptor);
    fds.events = POLLIN;

    struct netmap_ring* rxring = NULL;
    struct netmap_if* nifp = netmap_descriptor->nifp;

    // printf("Reading from fd %d thread id: %d", netmap_descriptor->fd, thread_number);

    for (;;) {
        // We will wait 1000 microseconds for retry, for infinite timeout please use -1
        int poll_result = poll(&fds, 1, 1000);

        if (poll_result == 0) {
            // printf("poll return 0 return code");
            continue;
        }

        if (poll_result == -1) {
            logger.error("Netmap plugin: poll failed with return code -1");
        }

        for (int i = netmap_descriptor->first_rx_ring; i <= netmap_descriptor->last_rx_ring; i++) {
            // printf("Check ring %d from thread %d", i, thread_number);
            rxring = NETMAP_RXRING(nifp, i);

            if (nm_ring_empty(rxring)) {
                continue;
            }

            receive_packets(rxring, thread_number);
        }

        // TODO: this code could add performance degradation
        // Add interruption point for correct toolkit shutdown
        // boost::this_thread::interruption_point();
    }

    // nm_close(netmap_descriptor);
}
```

```
void start_netmap_collection(process_packet_pointer func_ptr) {
    logger << log4cpp::Priority::INFO << "Netmap plugin started";
    netmap_process_func_ptr = func_ptr;

    num_cpus = sysconf(_SC_NPROCESSORS_ONLN);
    logger.info("We have %d cpus", num_cpus);

    std::string interfaces_list = "";

    if (configuration_map.count("interfaces") != 0) {
        interfaces_list = configuration_map["interfaces"];
    }

    if (configuration_map.count("netmap_sampling_ratio") != 0) {
        netmap_sampling_ratio = convert_string_to_integer(
    configuration_map["netmap_sampling_ratio"]);
    }

    if (configuration_map.count("
    netmap_read_packet_length_from_ip_header") != 0) {
        netmap_read_packet_length_from_ip_header = configuration_map["
    netmap_read_packet_length_from_ip_header"] == "on";
    }

    std::vector<std::string> interfaces_for_listen;
    boost::split(interfaces_for_listen, interfaces_list, boost::
    is_any_of(","), boost::token_compress_on);

    logger << log4cpp::Priority::INFO << "netmap will listen on " <<
    interfaces_for_listen.size() << " interfaces";

    // Thread group for all "master" processes
    boost::thread_group netmap_main_threads;

    for (std::vector<std::string>::iterator interface =
    interfaces_for_listen.begin();
        interface != interfaces_for_listen.end(); ++interface) {

        logger << log4cpp::Priority::INFO << "netmap will sniff
    interface: " << *interface;

        netmap_main_threads.add_thread( new boost::thread(receiver, *
    interface) );
    }

    netmap_main_threads.join_all();
}
```

Listing B.1: c script for netmap packet capture

Appendix C

Modified pcap based packet capturing and forwarding

```
\label{pcap}
/*******************************************************
* file:   pcap_plugin.c
* date:   2018-Feb-14 12:14:19 AM
* Author: Meklit Elfiyos
* Last Modified:2018-Feb-14 12:14:19 AM
*
* Description: fastnetmon pcap based \gls{pce}
*******************************************************/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <inttypes.h>

#include <map>
#include <string>

#include <pcap.h>
#include <netinet/if_ether.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>

// log4cpp logging facility
#include "log4cpp/Category.hh"
#include "log4cpp/Appender.hh"
#include "log4cpp/FileAppender.hh"
#include "log4cpp/OstreamAppender.hh"
#include "log4cpp/Layout.hh"
#include "log4cpp/BasicLayout.hh"
#include "log4cpp/PatternLayout.hh"
#include "log4cpp/Priority.hh"
```

```

#include "pcap_collector.h"


// Standard shift for type DLT_EN10MB, Ethernet
unsigned int DATA_SHIFT_VALUE = 14;

/* Complete list of ethertypes: http://en.wikipedia.org/wiki/EtherType
   */
/* This is the decimal equivalent of the VLAN tag's ether frame type
   */
#define VLAN_ETHERTYPE 0x8100
#define IP_ETHERTYPE 0x0800
#define IP6_ETHERTYPE 0x86dd
#define ARP_ETHERTYPE 0x0806
/* 802.1Q VLAN tags are 4 bytes long. */
#define VLAN_HDRLEN 4
#include "loop.c"
extern log4cpp::Category& logger;
extern std::map<std::string, std::string> configuration_map;

// This variable name should be uniq for every plugin!
process_packet_pointer pcap_process_func_ptr = NULL;

// Enlarge receive buffer for PCAP for minimize packet drops
unsigned int pcap_buffer_size_mbytes = 10;

// pcap handler, we want it as global variable beacuse it used in
   singnal handler
//pcap_t* descr = NULL;
pcap_t* descr_send=NULL;
//char errbuf[PCAP_ERRBUF_SIZE];
struct pcap_pkthdr hdr;
int set_buffer_size_res;
char devce[]="enp13s0";
// Prototypes
void parse_packet(u_char* user, struct pcap_pkthdr* packethdr, const
   u_char* packetptr);
void pcap_main_loop(const char* dev);

void start_pcap_collection(process_packet_pointer func_ptr) {
    logger << log4cpp::Priority::INFO << "Pcap plugin started";

    pcap_process_func_ptr = func_ptr;

    std::string interface_for_listening = "";

    if (configuration_map.count("interfaces") != 0) {
        interface_for_listening = configuration_map["interfaces"];
    }

```

```
    logger << log4cpp::Priority::INFO << "Pcap will sniff interface: " << interface_for_listening;

    pcap_main_loop(interface_for_listening.c_str());
}

void stop_pcap_collection() {
    // stop pcap loop
    pcap_breakloop(descr);
}

// We do not use this function now! It's buggy!
void parse_packet(u_char* user, struct pcap_pkthdr* packethdr, const u_char* packetptr) {
    struct ip* iphdr;
    struct tcphdr* tcphdr;
    struct udphdr* udphdr;

    struct ether_header* eptr; /* net/ethernet.h */
    eptr = (struct ether_header*)packetptr;

    if (ntohs(eptr->ether_type) == VLAN_ETHERTYPE) {
        // It's tagged traffic we should sjoft for 4 bytes for getting the data
        packetptr += DATA_SHIFT_VALUE + VLAN_HDRLEN;
    } else if (ntohs(eptr->ether_type) == IP_ETHERTYPE) {
        // Skip the datalink layer header and get the IP header fields.
        packetptr += DATA_SHIFT_VALUE;
    } else if (ntohs(eptr->ether_type) == IP6_ETHERTYPE or ntohs(eptr->ether_type) == ARP_ETHERTYPE) {
        // we know about it but does't not care now
    } else {
        // printf("Packet with non standard ethertype found: 0x%x\n", ntohs(eptr->ether_type));
    }

    iphdr = (struct ip*)packetptr;

    // src/dst UO is an in_addr, http://man7.org/linux/man-pages/man7/ip.7.html
    uint32_t src_ip = iphdr->ip_src.s_addr;
    uint32_t dst_ip = iphdr->ip_dst.s_addr;

    // The ntohs() function converts the unsigned short integer netshort from network byte order to
    // host byte order
    unsigned int packet_length = ntohs(iphdr->ip_len);

    simple_packet current_packet;

    // Advance to the transport layer header then parse and display
```

```c
    // the fields based on the type of hearder: tcp, udp or icmp
    packetptr += 4 * iphdr->ip_hl;
    switch (iphdr->ip_p) {
    case IPPROTO_TCP:
        tcphdr = (struct tcphdr*)packetptr;

#if defined(__FreeBSD__) || defined(__APPLE__) || defined(__DragonFly__)
        current_packet.source_port = ntohs(tcphdr->th_sport);
#else
        current_packet.source_port = ntohs(tcphdr->source);
#endif

#if defined(__FreeBSD__) || defined(__APPLE__) || defined(__DragonFly__)
        current_packet.destination_port = ntohs(tcphdr->th_dport);
#else
        current_packet.destination_port = ntohs(tcphdr->dest);
#endif
        break;
    case IPPROTO_UDP:
        udphdr = (struct udphdr*)packetptr;

#if defined(__FreeBSD__) || defined(__APPLE__) || defined(__DragonFly__)
        current_packet.source_port = ntohs(udphdr->uh_sport);
#else
        current_packet.source_port = ntohs(udphdr->source);
#endif

#if defined(__FreeBSD__) || defined(__APPLE__) || defined(__DragonFly__)
        current_packet.destination_port = ntohs(udphdr->uh_dport);
#else
        current_packet.destination_port = ntohs(udphdr->dest);
#endif
        break;
    case IPPROTO_ICMP:
        // there are no port for ICMP
        current_packet.source_port = 0;
        current_packet.destination_port = 0;
        break;
    }

    current_packet.protocol = iphdr->ip_p;
    current_packet.src_ip = src_ip;
    current_packet.dst_ip = dst_ip;
    current_packet.length = packet_length;

    // Do packet processing
    pcap_process_func_ptr(current_packet);
  //==========================================================
```

```
  /* open device for reading */
  pcap_sendpacket(handler, packetptr, packethdr->len);
  //==========================================================================
}

void pcap_main_loop(const char* dev) {
    char errbuf[PCAP_ERRBUF_SIZE];
    /* open device for reading in promiscuous mode */
    int promisc = 1;

    bpf_u_int32 maskp; /* subnet mask */
    bpf_u_int32 netp; /* ip */

    logger << log4cpp::Priority::INFO << "Start listening on " << dev;

    /* Get the network address and mask */
    pcap_lookupnet(dev, &netp, &maskp, errbuf);

    descr = pcap_create(dev, errbuf);

    if (descr == NULL) {
        logger << log4cpp::Priority::ERROR << "pcap_create was failed with error: " << errbuf;
        exit(0);
    }

    // Setting up 1MB buffer
    set_buffer_size_res = pcap_set_buffer_size(descr, pcap_buffer_size_mbytes * 1024 * 1024);
    if (set_buffer_size_res != 0) {
        if (set_buffer_size_res == PCAP_ERROR_ACTIVATED) {
            logger << log4cpp::Priority::ERROR
                   << "Can't set buffer size because pcap already activated\n";
            exit(1);
        } else {
            logger << log4cpp::Priority::ERROR << "Can't set buffer size due to error: " << set_buffer_size_res;
            exit(1);
        }
    }

    if (pcap_set_promisc(descr, promisc) != 0) {
        logger << log4cpp::Priority::ERROR << "Can't activate promisc mode for interface: " << dev;
        exit(1);
    }
  pcap_setdirection(descr, PCAP_D_IN);

   if (pcap_activate(descr) != 0) {
        logger << log4cpp::Priority::ERROR << "Call pcap_activate was failed: " << pcap_geterr(descr);
```

```
        exit(1);
    }
//======================================== filter ========================
  char filter_exp[255] = " not (dst host 10.10.10.172 and dst port
    2323 and tcp[tcpflags]==tcp-syn) ";
  struct bpf_program filter;
  bpf_u_int32 subnet_mask, ip;
  if (pcap_compile(descr, &filter, filter_exp, 0, ip) == -1) {
        printf("Bad filter - %s\n", pcap_geterr(descr));

    }
    if (pcap_setfilter(descr, &filter) == -1) {
        printf("Error setting filter - %s\n", pcap_geterr(descr));

    }
    //======================================================================
   // man pcap-linktype
    int link_layer_header_type = pcap_datalink(descr);

    if (link_layer_header_type == DLT_EN10MB) {
        DATA_SHIFT_VALUE = 14;
    } else if (link_layer_header_type == DLT_LINUX_SLL) {
        DATA_SHIFT_VALUE = 16;
    } else {
        logger << log4cpp::Priority::INFO << "We did not support link
    type:" << link_layer_header_type;
        exit(0);
    }
  if(dev == NULL)
    { printf("%s\n",errbuf); exit(1); }

   if (descr == NULL) {
        logger << log4cpp::Priority::ERROR << "pcap_create was failed
    with error: " << errbuf;
        exit(0);
    }
  handler = pcap_create("enp0s25", errbuf);
    // Setting up 1MB buffer
    set_buffer_size_res = pcap_set_buffer_size(handler,
    pcap_buffer_size_mbytes * 1024 * 1024);
    if (set_buffer_size_res != 0) {
        if (set_buffer_size_res == PCAP_ERROR_ACTIVATED) {
            logger << log4cpp::Priority::ERROR
                   << "Can't set buffer size because pcap already
    activated\n";
            exit(1);
        } else {
            logger << log4cpp::Priority::ERROR << "Can't set buffer
    size due to error: " << set_buffer_size_res;
            exit(1);
        }
    }
```

```
    if (pcap_set_promisc(handler, promisc) != 0) {
        logger << log4cpp::Priority::ERROR << "Can't activate promisc mode for interface: " << dev;
        exit(1);
    }
  pcap_setdirection(handler, PCAP_D_IN);

    if (pcap_activate(handler) != 0) {
        logger << log4cpp::Priority::ERROR << "Call pcap_activate was failed: " << pcap_geterr(descr);
        exit(1);
    }

    // man pcap-linktype
    link_layer_header_type = pcap_datalink(handler);

    if (link_layer_header_type == DLT_EN10MB) {
        DATA_SHIFT_VALUE = 14;
    } else if (link_layer_header_type == DLT_LINUX_SLL) {
        DATA_SHIFT_VALUE = 16;
    } else {
        logger << log4cpp::Priority::INFO << "We did not support link type:" << link_layer_header_type;
        exit(0);
    }
     if (handler == NULL) {
        logger << log4cpp::Priority::ERROR << "pcap_create was failed with error: " << errbuf;
        exit(0);
    }
    handler = pcap_open_live("enp0s25",pkt_sizes,1,-1,errbuf);

    pcap_loop(descr, -1, (pcap_handler)parse_packet, NULL);
  }

std::string get_pcap_stats() {
    std::stringstream output_buffer;

    struct pcap_stat current_pcap_stats;
    if (pcap_stats(descr, &current_pcap_stats) == 0) {
        output_buffer << "PCAP statistics"
                      << "\n"
                      << "Received packets: " << current_pcap_stats.ps_recv << "\n"
                      << "Dropped packets: " << current_pcap_stats.ps_drop << " ("
                      << int((double)current_pcap_stats.ps_drop / current_pcap_stats.ps_recv * 100) << "%)"
                      << "\n"
                      << "Dropped by driver or interface: " << current_pcap_stats.ps_ifdrop << "\n";
```

```
    }

    return output_buffer.str();
}
```

Listing C.1: c script for pcap based fastnetmon packet capture engine

```
\label{outgoing}
/*****************************************************
* file:   outgoing.c
* date:   2018-Feb-14 12:14:19 AM
* Author: Meklit Elfiyos
* Last Modified:2018-Feb-14 12:14:19 AM
*
* Description: captures from one interface and sends to another
    interface
*****************************************************/

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
int pkt_sizes=(10* 1024 * 1024);
/* Ethernet header */
  struct sniff_ethernet {
    u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
    u_char ether_shost[ETHER_ADDR_LEN]; /* Source host address */
    u_short ether_type; /* IP? ARP? RARP? etc */
  };

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header
    length >> 2 */
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits
    */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
```

```
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
}*ip=NULL;
 int i;
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];

    const u_char *packet;
    struct pcap_pkthdr hdr;      /* pcap.h */
    struct ether_header *eptr;   /* net/ethernet.h */
  const struct sniff_ethernet *ethernet; /* The ethernet header */
 pcap_t* handler_send=NULL;
 pcap_t* handler=NULL;
/* callback function that is passed to pcap_loop(..) and called each time
 * a packet is recieved
 */
 unsigned long long user_mac_in_int(const struct sniff_ethernet *ethernad)
{
  unsigned long long user_mac=0,tmp=0;

  tmp          =ethernad->ether_shost[0];
  user_mac=tmp;
  tmp          =ethernad->ether_shost[1];
  user_mac<<=8;
  user_mac+=tmp;
  tmp          =ethernad->ether_shost[2];
  user_mac<<=8;
  user_mac+=tmp;
  tmp          =ethernad->ether_shost[3];
  user_mac<<=8;
  user_mac+=tmp;
  tmp          =ethernad->ether_shost[4];
  user_mac<<=8;
  user_mac+=tmp;
  tmp          =ethernad->ether_shost[5];
  user_mac<<=8;
  user_mac+=tmp;
  return user_mac;
}
//=================================================
void my_callback(u_char *useless,const struct pcap_pkthdr* pkthdr, const u_char*
        packet)
{
  dev = "eth1"; //pcap_lookupdev(errbuf);
    if(handler_send== NULL)
  {
    handler_send = pcap_open_live(dev,pkt_sizes,1,-1,errbuf);
    if(handler_send == NULL)
    {
    printf("pcap_open_live(): %s\n",errbuf); exit(1); }
```

```
  }
    if (pcap_sendpacket(handler_send, packet, pkthdr->caplen) == 0) {

  }
}

int main(int argc,char **argv)
{
    dev = "eth1";
  if(dev == NULL)
    { printf("%s\n",errbuf); exit(1); }
    /* open device for reading */

    handler = pcap_open_live(dev,pkt_sizes,1,-1,errbuf);
  pcap_setdirection(handler, PCAP_D_IN);
    if(handler == NULL)
    {
    printf("pcap_open_live(): %s\n",errbuf); exit(1);
  }

    pcap_loop(handler,-1,my_callback,NULL);
 return 0;
}
```

Listing C.2: c script for outgoing traffic