Ganesh Sharma

Evaluating the performance of Netfilter architecture in Private Realm Gateway

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 13.08.2018

Thesis supervisor:

Prof. Raimo Kantola

Thesis advisor:

Jesús Llorente Santos

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

Abstract of the
Master's Thesis

Author: Ganesh Sharma

Title: Evaluating the performance of Netfilter architecture in Private Realm Gateway

Date: 13.08.2018 Language: English Number of pages: 6+69

Department of Communications and Networking

Professorship: Networking Technology Code: COM.thes

Supervisor: Prof. Raimo Kantola

Advisor: Jesús Llorente Santos

Network address translation (NAT) was introduced to decelerate the IPv4 ad-dresses depletion through separation of a network into the public and private realm. The hosts in a private network connect to the public Internet by sharing a pool of public IP addresses, and NAT acts as a gateway between the public and the private networks. Although NAT alleviates the problem of addresses depletion, it leads to a reachability problem as NAT would generally block any outside connections to the private network from the Internet.

This thesis examines a new concept called Private Realm Gateway (PRGW) which is developed to overcome the shortcoming of NAT. PRGW imitates the NAT functionality and allows the inbound connections initiated from the public networks towards a private realm via the Circular Pool of Public Addresses (CPPA). PRGW provides interoperability between the legacy IP network and hosts in the private networks and vice-versa, using pre-existing TCP/IP protocols and applications.

PRGW has been implemented on top of Linux operating system, and therefore, the primary approach in this thesis is to evaluate the forwarding performance of Linux kernel networking (Netfilter subsystem), as well as inspect the possible performance tuning methods to achieve higher packets processing rates.

The performance of Netfilter is evaluated by offering heavy traffic load to measure packet forwarding capability, memory usage by IP traffic as well as overloading the CPU process. In addition, the stateful mechanism for packet filtering and NAT routing was evaluated using appropriate iptables lookup and packets traversing through different chains. When conducting the various tests, by adjusting different parameters in Linux Netfilter subsystem revels that the PRGW can be deployed over the Linux architecture.

Keywords: PRGW, CES, NAT, Connection tracking, Netfilter, iptable, IP

Preface

I would like to take this moment to express my sincere gratitude to Professor Raimo Kantola for giving me an opportunity to work with him and his team.

I am grateful to Jesus Llorente Santos of his valuable advice and guidance he has given me throughout the duration of this thesis work. His sincere guidance and support for my research work and his suggestion has shaped my thesis and his critical comments helped me improve the quality of my work.

I would also like to thanks my friends for their support and motivation during my thesis work and during my difficult times.

Last but not the least, I would like to thank my parents and my brothers for their immense love and encouragement during all these years.

Otaniemi, 13.08.2018

Ganesh Sharma

Contents

Abbreviations

AH	Authentication Header
API	Application Programming Interface
ARP	Address Resolution Protocol
CES	Customer Edge Switching
CETP	Customer Edge Traversal Protocol
CIDR	Classless Inter-Domain Routing
CPPA	Circular Pool of IP Addresses
CN	Customer Network
CPU	Central Processing Unit
DNS	Domain Name System
DNAT	Destination Network Addresses Translation
FQDN	Fully Qualified Domain Name
FTP	File Transfer Protocol
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPS	Intrusion Prevention System
MAC	Media Access Control
MB	MegaBytes
MSL	Maximum Segment Length
NAT	Network Address Translation
NAPTR	Name Authority Pointer
NUMA	Non-uniform Memory Access
OS	Operating System
OSI	Open Systems Interconnection
PRGW	Private Realm Gateway
RAM	Random Access Memory
RFC	Request for Comments
RLOC	Routing Locator
SCTP	Stream Control Transmission Protocol
SPN	Service Provider Network
SNAT	Source Network Address Translation
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TTL	Time To Live
TOS	Type Of Service
UDP	User Datagram Protocol
UN	User Network

1 Introduction

Since the dawn of the Internet, it is growing exponentially and transforming the means of communication every day. The Internet has become the underlying communication mechanism for information sharing, collaboration and interaction between individuals regardless of the geographical location. Behind the massive structure of the Internet, TCP/IP has played the most significant role for expansion of communications and connecting the vast numbers of network devices on the Internet. Despite the tremendous success of the Internet, today, the Internet is facing a problem of address exhaustion after all the number of IP addresses is not unlimited. To resolve the address exhaustion problem, long-term, as well as short-term solutions, are being developed.

IPv6 being the long-term solution has yet to be deployed across the Internet. Therefore, until IPv6 is ready and take over the demand for IP addresses, short-term solution, for instance, CIDR, RFC 1918 addresses and NAT have been compensating for the problem. In order to address the issue of IP address depletion, NAT is being extensively implemented across the Internet. A NAT is a device that translates the private IP addresses to the public addresses and vice versa. In a NAT framework, a single node acts as a midpoint between a private network and the public network. As a result, an individual or a pool of unique IP addresses represent a larger group of hosts in the global IP network.

Furthermore, the NAT device acts as a firewall, blocking the incoming connections to the private network from the public realm. Consequently, a host from the public Internet cannot initiate communication with a host in the private network. [1] Thus, the deployment of NAT causes the loss of end-to-end IP reachability.

To overcome the reachability problems, Professor Raimo Kantola, at the Department of Communications and Networking of Aalto University, proposed a new technology called "Customer Edge Switching (CES)." CES is based on the transition from the end-to-end principle to the trust-to-trust principle and replacement of NAT. CES allows a private host to be globally reachable across the public Internet by conducting trust and policy negotiation between the communicating edge nodes. In order to establish trust, both sites should be running CES at their edge, and both the ends have to be identified through globally unique domains names. [3]

Moreover, communication from the legacy IP networks to the private hosts can be carried out via Realm Gateway also known as Private Realm Gateway (PRGW). Furthermore, in order to be able to deploy CES one network at a time, the CES and the PRGW functionality can be introduced in a single edge node. Alternatively, the PRGW can be implemented as a standalone solution to overcome the weakness of the NAT transversal that is needed at present. [2]

1.1 Motivation

Private Realm Gateway(PRGW) is developed to overcome the shortcomings of NAT. PRGW depict the functionality of NAT solutions and allows unilaterally initiated inbound connections from the public networks towards the private hosts via the Circular Pool of Public Addresses (CPPA). The PRGW architecture has been implemented over the Linux operating system. Therefore, the primary motive of this paper is to evaluate the forwarding performance of Linux kernel networking (Netfilter subsystem) and inspect the possible performance tuning methods to achieve higher packet processing rates.

1.2 Objectives

The objective of this thesis is to find the possible bottlenecks of Linux kernel in terms of packets processing and viability of deploying Private Realm Gateway (PRGW) in terms of those bottlenecks.

In order to achieve the research objective, the following research questions were identified:

– What tools and test frameworks can be used for testing Realm Gateway?

– How can the selected testing environment be implemented in end-to-end testing?

– How would the possible bottlenecks in packet processing in the Linux influence the performance of PRGW?

– What could be proposed for optimization of possible bottlenecks and how?

1.3 Thesis Structure

The thesis has been sub-divided into five chapters; Chapter 2 provides a theoretical background of the literature review associated with the topics of the thesis. In addition, the test suite development process has been described in Chapter 3 and detailed description of results, evaluation and discussion are outlined in Chapter 4, followed by conclusions and discussion in Chapter 5.

2 Background

This chapter focuses on discussing some fundamental concepts and definitions to understand the key notion of this thesis. The Section 2.1, describes the necessary background information relevant to the Network Address Translation (NAT). In Section 2.2 and 2.3, the technology underlying the Customer Edge Switching (CES) and the Private Realm Gateway (PRGW) are discussed. Moreover, the Section 2.4 and 2.5 discusses the Linux Netfilter architecture and Connection tracking on which this thesis is primarily focused.

2.1 Network Address Translation (NAT)

Network address translation (NAT), was designed as a short-term solution for solving the IPv4 address depletion until long-term solutions are operational. In a NAT framework, a single node acts as a midpoint between the public and the private networks. And consequently, NAT binds the private IP addresses with the globally routable IP addresses and vice versa, to provide the forwarding functionality for IP packets traversing between the private and the public networks. [8]

NAT uses RFC 1918 private addresses inside the private networks. These private IP addresses are then translated to globally unique public IP addresses for connecting to an outside network. In a NAT architecture, a single or a pool of IP addresses represent the entire private network on the Internet. Therefore, a NAT device can connect two networks and translate a private IP (not globally unique) addresses in the internal network to globally unique routable IP addresses before IP packets are forwarded. [9]

NAT is classified into static and dynamic NAT translation. In static translation, a single private IP address is mapped to a unique public IP address. The static translation is not efficient as this solution cannot provide a mechanism for solving the IPv4 address depletion problem. This solution can be used for servers and devices whose IP addresses are fixed. In dynamic translation, however, NAT randomly translates the provided addresses using its pool of public IP addresses. When the last session using an address binding is terminated, NAT would free the binding so that the global address is recycled for the later use. NAT devices maintain a table, called NAT table, for keeping track of sent IP packets and incoming IP packets. NAT devices maintain a state for every connection with outside networks and any outside connection that does not match any state will be denied. [7]

A NAT device maintains a state for mapping the translated IP packets for every connection. A mapping is dynamically allocated for the connection initiated internally and potentially reused for the specific subsequent connections. A host from a private network initiates a connection through NAT by sending the first packet. NAT allocates (or reuses) mapping for a connection where the mapping holds a tuple of IP addresses and the port used for translation of all the IP packets for that connection. [6]

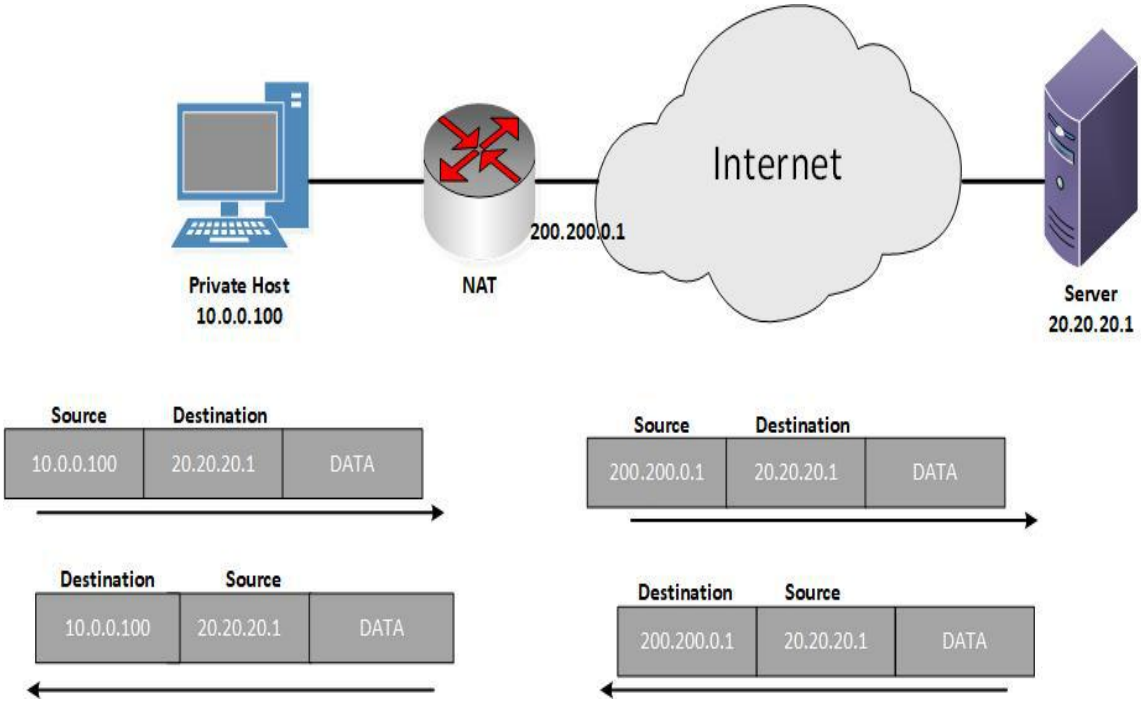


Figure 1: NAT translation

The detail working mechanism of NAT is shown in Figure 1. The diagram illustrates the working principle of NAT, where a host (IP: **10.0.0.100**) in a private network is communicating with the server (IP: **20.20.20.1**) in the public domain. The host creates an IP packet with the destination IP field to be the IP address of the server, i.e., **20.20.20.1**, while source address to be the host itself, i.e., **10.0.0.100**. Then, the private host sends the IP packet to the default gateway, i.e., the NAT router. When the default gateway receives the IP packet, it would look into IP header and see a private address, which cannot be routed in the Internet. Therefore, the private address is translated to the outbound public address in this case **200.200.0.1**, which is globally unique and can be routed in the Internet.

Now a new packet is created with the source from the NAT router interface and the destination to be the server (i.e., **200.200.0.1 > 20.20.20.1**). The translation information is stored in a NAT table, and when receiving response traffic from a server, the NAT table is examined to find the matching entry. If found, the NAT router replaces the destination again with the IP address of the private host (**20.20.20.1 > 10.0.0.100**) and forwards the packet.

As the example (above) illustrate, NAT devices represents an entire private net-work via only one or a pool of IP addresses in the global network. This ability provides additional security by effectively hiding the entire internal network behind the one address. However, a NAT device causes a reachability problem by hiding a private network from the public Internet. [8] [9]

Although NAT allows the public IP addresses to be shared by a large number of hosts in a private network, its deployment leads to the incoming reachability problem. The reachability problem restricts an individual host from being reachable via the Internet and from accepting connections via the public network or a different private network.

2.2 Customer Edge Switching (CES)

Customer Edge Switching (CES) is a new technology developed in order to replace the NAT. CES is a new type of firewall based on the principle of trust-to-trust be-tween the public and the different private networks. CES connects two components namely: Customer/User Network (CN/UN) that can use the provided IP addresses for the hosts, while Service Provider Network (SPN) uses the globally unique IP addresses. This separation of different networks provides complete isolation and transparency, by creating the possibility to use new technologies and protocols in different network domains. For example, a core network could be running IPv4, IPv6, IP/MPLS or Ethernet independently from the technology used in a CN. In addition, the core network provides Directory Services (DS) for domain resolution, such as DNS. [3]

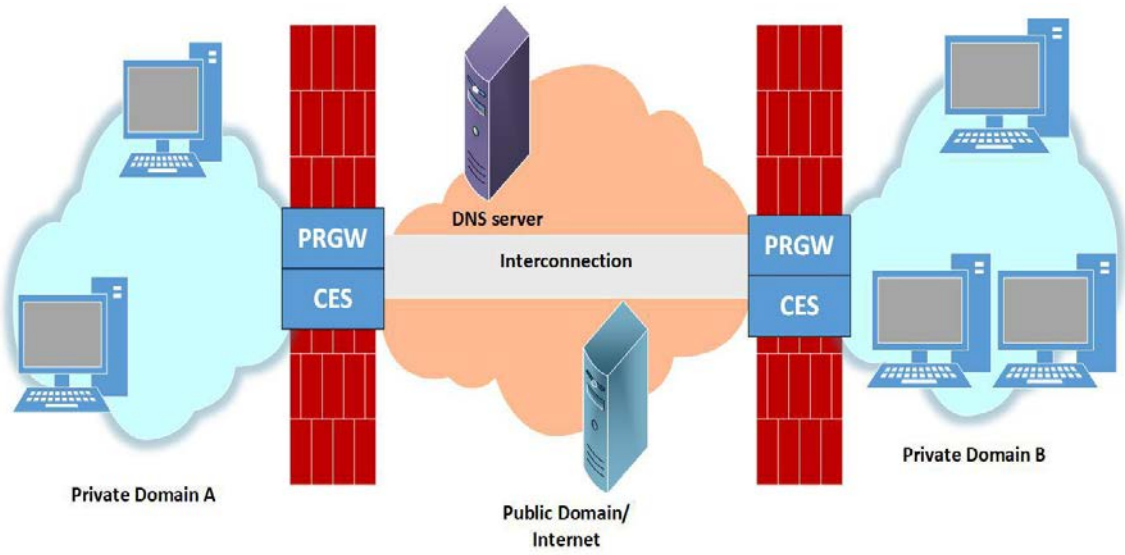


Figure 2: CES Architecture

The architecture of CN consists of private end hosts while SPN consists of the core networks that interconnects and routes traffic between the different customer networks. CES at the edge of CN and SPN networks operates as the stateful firewall and allows or denies the traffic based on the policy. Furthermore, CES allows a private host to be globally reachable across the public Internet by establishing trust and policy negotiations between communicating edge nodes. In order to establish the trust, both sites should be running CES at the edge and all hosts should be identified through a

globally unique fully qualified domain name (FQDN). The CES architecture is deployed on top of the existing legacy Internet framework using the current IP addresses scheme, as well as extensively using DNS for global reachability. The hosts residing in a private domain are identified by using their FQDN instead of unique IP addresses, while IP or MAC addresses are used as routing locators(RLOCs).[1]

A DNS name resolution triggers every communication in the CES. A name resolution is the first step in creating a valid connection state in the CES. CES uses Name Authority Pointer (NAPTR) queries in order to provide extensibility and better support for abstract identifiers during communication with another CES. CES uses the circular pool of private IP addresses for addressing the end-host and the public address to identify a host uniquely in the public domain. Furthermore, CES implements both inbound and the outbound policy negotiation for creating a state and allows the packets to flow or be discarded based on the policy. Thus, the final 'allow' or 'drop' decision to the inbound packets is determined only after a policy negotiation. Therefore, the policy negotiation prevents unwanted traffic towards the private hosts. [1]

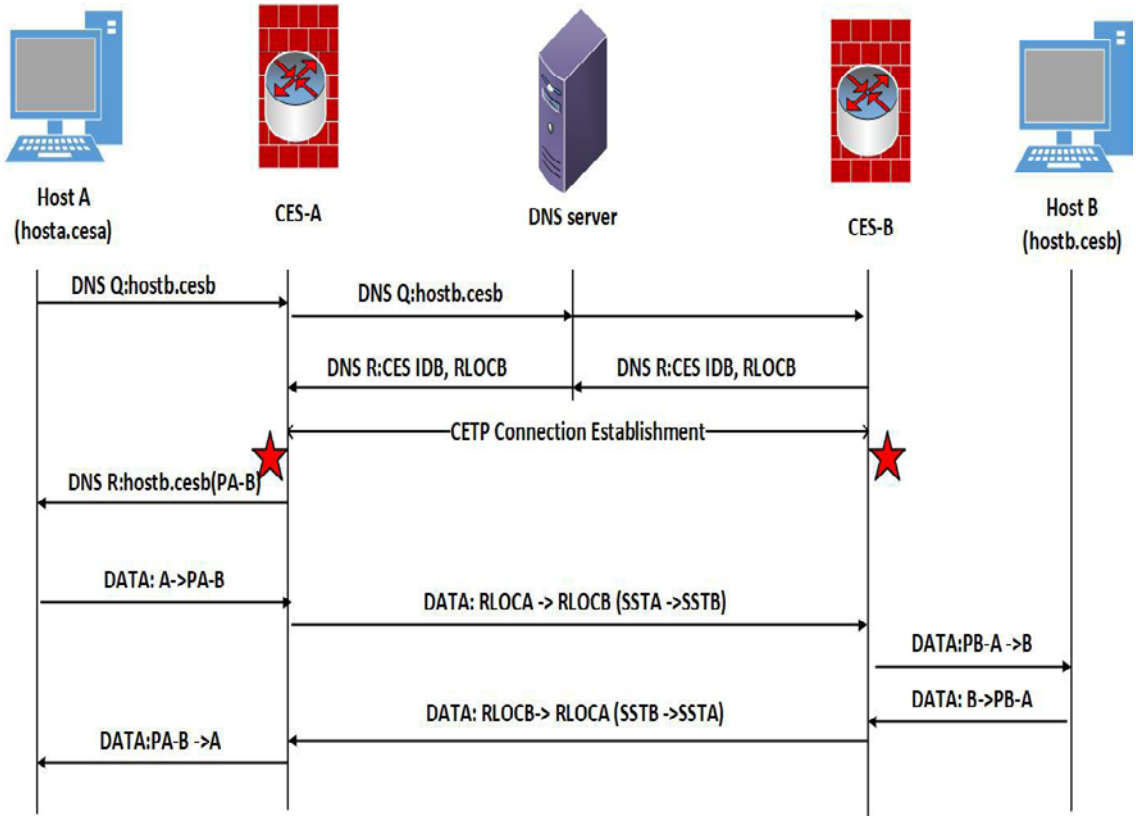


Figure 3: CES to CES Communication

Table 1: Notations

Legends	Definitions
A	Private IP for Host-A
B	Private IP for Host-B
R	Public IP for Host-A
PA-B	Proxy -address representing Host-B to Host-A
PB-A	Proxy -address representing Host-A to Host-B
IDA	ID of Host-A
IDB	ID of Host-B
RLOCA	Routing Locator of CES-A
RLOCB	Routing Locator of CES-B
SSTA	Session Tag for session initiated in CES-A
SSTB	Session Tag for session initiated in CES-B
	Creation of waiting state on incoming DNS query

Figure 3 shows a CES to CES communication. The scenario consists of two different CES networks: CES-A and CES-B, which are interconnected by a service provider network (SPN). The Host-A performs a DNS name resolution for destination host, i.e., Host-B to establish a session. Then, the Host-A sends a DNS query for hostb.cesb to CES-A, at this point the CES-A initiate a NAPTR resolution to DNS. In turn, DNS response conveys the information about the host-B ID and a routing locator for the CES-B. Next, CES-A initiates a policy negotiation with CES-B. When the policy match is satisfied for the Host B, CES-A sends back DNS response to the Host A. The response contains the allocated private IP address. As a result, the Host-A sends the data packets to a given proxy-address where CES-A will process and forward them accordingly towards the destination CES-B.

Similarly, the destination CES-B carries out a connection establishment procedure by a host admission policy. A successful connection establishment creates a connection state in each CES device, where CES represents a remote host locally using a proxy address. Both hosts exchange data packets using the respective proxy address.

2.3 Realm gateway

Private Realm Gateway (PRGW) is designed to provide interoperability between the legacy IP networks and the host in a private network and vice-versa, by using the existing TCP/IP protocols and applications. Besides, PRGW is a component of CES that can be implemented in standalone edge devices as well as integrated into CES. PRGW provides inward reachability from legacy Internet towards a host that resides behind a CES.

PRGW aims to replace the NAT devices at edge networks; in turn, reducing the problem of IPv4 addresses depletion. PRGW use a pool of addresses at the public side allowing the inbound connection towards the private networks. In addition, the communication between the legacy IP host and a host in the private domain would be carried out identically as NAT by sharing a single/pool of public addresses. [2][4]

2.3.1 Realm Gateway Architecture

PRGW proposed an architectural solution to overcome the drawbacks of the classical NAT solution. PRGW mirrors the functionality of conventional NAT solution by allowing the IP devices residing in a private realm to be able to communicate to the public networks by sharing a single or pool of IP addresses. Contrary, to the NAT, PRGW allows unilaterally initiated inbound connections from the public net-works towards the private host via the circular pool of public addresses (CPPA). In addition, PRGW, distinguishes networks into the public realm and a private realm as shown in the Figure 4.

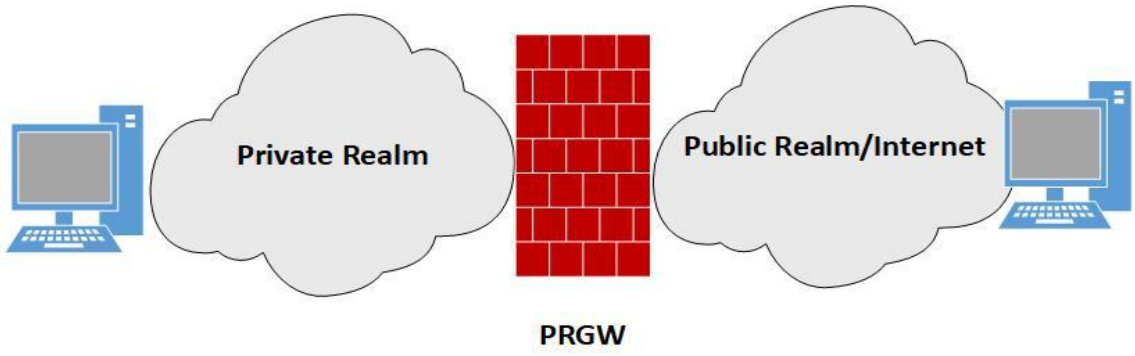


Figure 4: Realm Gateway

Figure 5 illustrates a typical PRGW communication between a host-A in private realm and public host-B in public domain. The public host-B sends DNS query on the FQDN of a private host-A, i.e., hosta.foo. Upon receiving the DNS query, PRGW allocates a public IP address from its pool to represent the host-A on the Internet. At this point, temporary half connections state has been created that allows inbound data flow/initiation to private host-A.

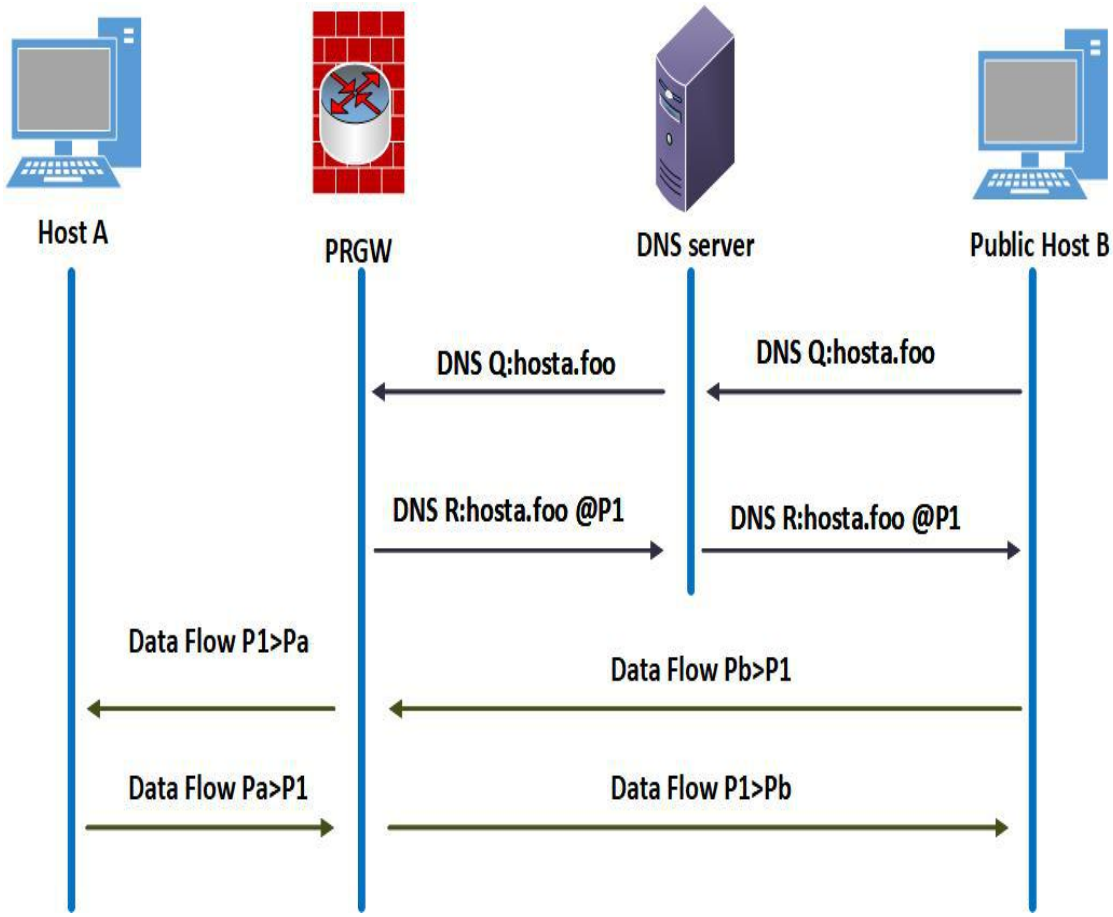


Figure 5: PRGW Communication

2.3.2 Inbound communication

The host residing in the public network sends a DNS query for hosta.foo to its name server. The DNS server relays the message to a PRGW and upon receiving the DNS query, host-A reserves an address from the circular pool. The DNS response is sent back to the public host containing the reserved IP address.

When public host-B receives the DNS response, it sends data packets to the reserved address. Then the packet is forwarded to the private host after performing public-to-private address translation. At this point, PRGW creates a full five-tuple connection entry. Similarly, the response from the private host is sent back to the public host after performing private-to-public address translation at PRGW. [4]

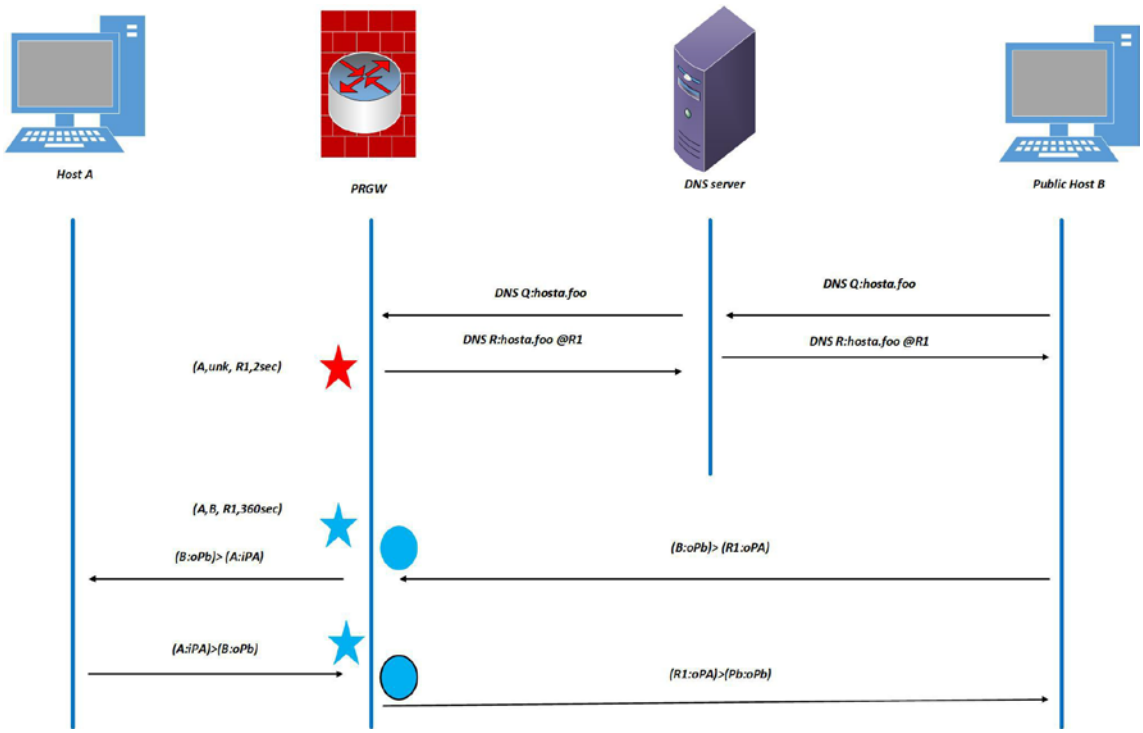


Figure 6: Inbound Communication

Table 2: Notations

Legends	Definitions
A	Private IP for Host-A
B	Private IP for Host-B
R1	Public IP for Host-A
iPA	Local port of Host-A
oPB	Local port of Host-B
	Creation of waiting state on incoming DNS query
	Creation of connection state
	Inbound translation modifies the destination IP address and port with the private host information
	Outbound translation modifies the source IP address and port with the public host information

2.3.3 Outbound Communication

The host from a private network initiates a connection to the public host as shown in the Figure 7. The outbound connection to the public host would be translated into the public address similar to classical NAT, and PRGW would maintain a connection

state. When the response from the public host is received PRGW will look up its connection states, and if a connection is valid, inbound translation of packet would be carried out, and the packet will be delivered to the private host. However, during the inbound connection from the public host, the PRGW would use the circular pool of public addresses (CPPA) for accepting the incoming connections. [2]

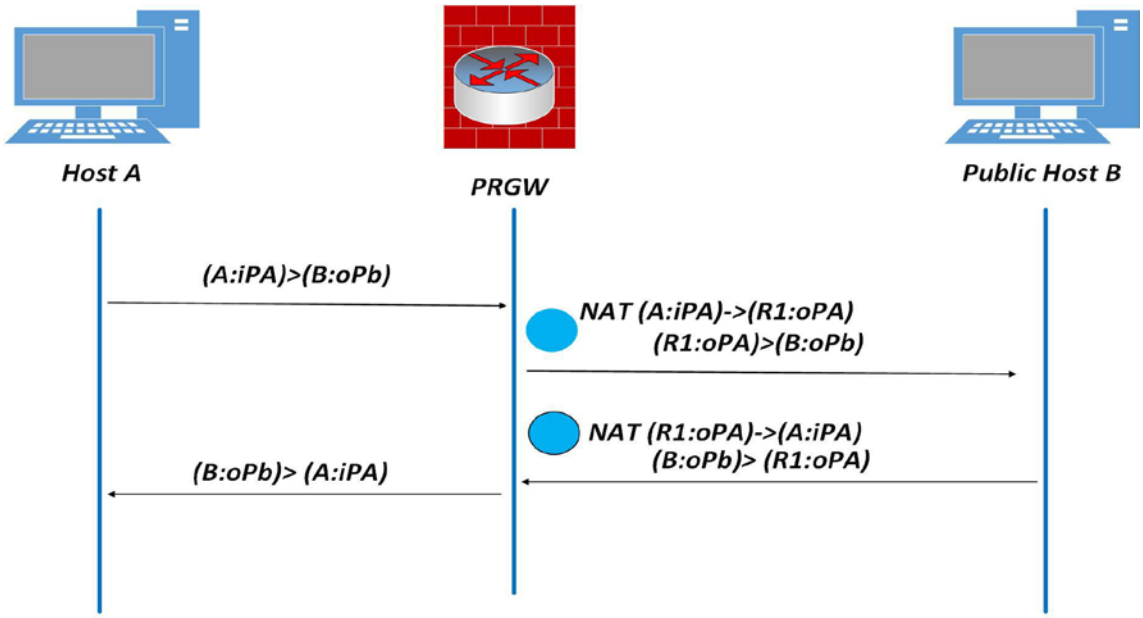


Figure 7: PRGW Outbound communication

Table 3: Notations

Legends	**Definitions**
A	Private IP for Host-A
R1	Public IP for Host-A
B	Public IP for Host-B
iPA	Local port of Host-A
oPA	Public port of Host-A
oPb	Public port of Host-B
●	Inbound translation modifies the source IP address and port with the public host information
●	Outbound translation modifies the destination IP address and port with the private host information

2.4 Netfilter

Netfilter is a network subsystem that offers user-space tools to control the packet forwarding functionality in the Linux kernel. Netfilter provides a mechanism for packet mangling, packet filtering, network address translation as well as port translation implemented by the kernel networking stack. Furthermore, Netfilter is a set of different intercepted function calls known as hooks inside Linux kernel. [10] These hooks will be triggered/callback when an IP packet enters into the Linux networking stack and depending on the packet property, one or the other hook is called. Netfilter hook controls the traffic flow inside the kernel networking stack. There are five Netfilter hooks that will be triggered/callback when the IP packet enters into the Linux networking stack. Table 4 describes the different hooks and their functionality.[13]

Table 4: Netfilter Hooks

Hook	**Definations**
NF_IP_PRE_ROUTING	All the incoming IP packets will trigger this hook. Further, this hook is processed before any routing decisions take place regarding where to send the packet.
NF_IP_LOCAL_IN	When the incoming IP packets is destined for the local system, this hook will be triggered after the routing of IP packets has been done.
NF_IP_FORWARD	This hook is triggered after an incoming packets has been routed, and when IP packet is to be routed to another host
NF_IP_LOCAL_OUT	Any locally generated outbound IP packets triggered this hook.
NF_IP_POST_ROUTING	This hook is triggered by any outgoing or forwarded IP packets after routing has taken place and just before being put on the wire.

Therefore, all the IP packets, both the incoming and the outgoing through the network device have to be processed through Netfilter network subsystem. [10]

2.4.1 Netfilter chains

Netfilter chains allow the administrative control of the IP packet delivery and the packet flow within the networking stack. These chains will be triggered when IP packet registers with netfilter hooks. Netfilter consist of five chains namely PREROUTING, INPUT, FORWARD, OUTPUT and finally POSTROUTING. These chains will be triggered sequentially throughout packet flow, for example, every

packet without any exceptions will first hit the PREROUTING chain. In the PREROUTING chain, the packet will be processed through different tables, for instance, RAW table, MANGLE table and then, NAT table.

Soon after the PREROUTING chain, the routing of the IP packet takes place, where the packet is sent either to INPUT chain or FORWARD chain depending on the final destination of the packet. The packet will hit the INPUT chain if the IP packet is destined for the local system. And the packet will hit FORWARD chain if the packet is destined for a remote system. In both situations, filtering as well mangling of the packet is carried out. In addition, if the local system has generated the IP packet, the packet will hit the OUTPUT chain. And finally, POSTROUTING chain is triggered, where mangle and source NAT is carried out.

Figure 8 illustrates the packet flow inside the Netfilter subsystem. [17]

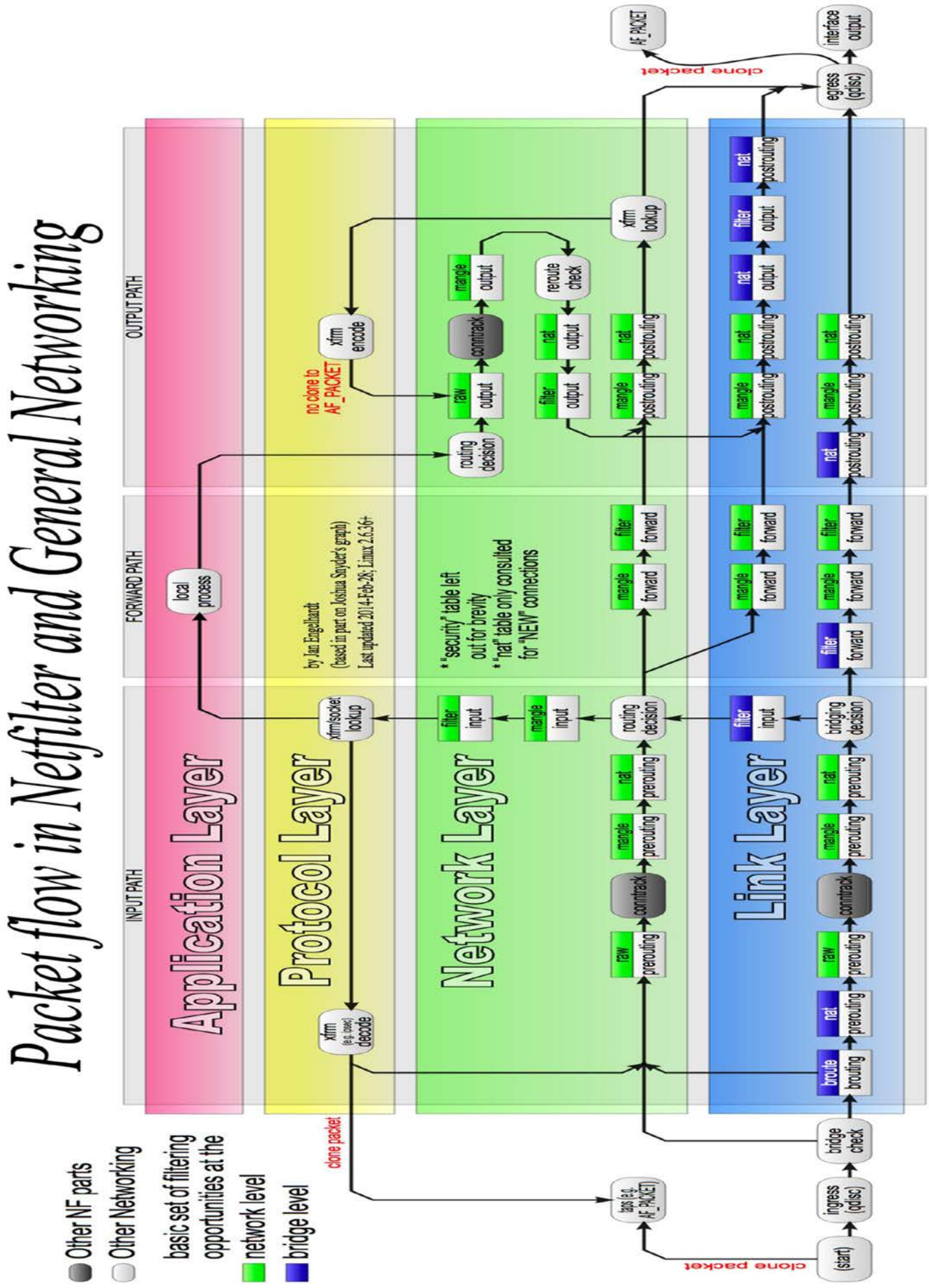


Figure 8: Packet flow in Netfilter and General networking in Linux

2.4.2 Netfilter packet flow

When the physical interface receives an IP packet, NIC driver passes the packet to the kernel. Next, the packet goes through a series of different steps in the kernel networking stack to be appropriately processed and delivered to the correct application/system. All the IP packets traversing via networking stack firs register in PREROUTING chain and its corresponding table. In the PREROUTING chain, firstly, the packet will hit the RAW table. The RAW is primarily used for configuring exemptions from connection tracking. Therefore, the IP packets that are marked as NOTRACK in RAW table will not appear in connection tracking. [16] And then after the RAW table packets arrive at the connection tracking, where the connection state is stored.

Soon after connection tracking, packets will go through the MANGLE table. The MANGLE table is used for altering packet fields, for instances TOS (Type of services), DSCP, ECN, MARK, IPMARK, and ROUTE. Next, to the MANGLE table, NAT table is triggered, where the destination address of the IP packet is changed, i.e., DNAT (Destination NAT) and REDIRECT, BALANCE is also carried out. On most IP packets, only the destination address of IP packet will be modified. Finally, the IP packet hits the routing table, in which a decision is made to send the IP packet either to the INPUT chain or the FORWARD chain based on the destination of IP packets. [13]

– IP packets destined for the local process/applications.

The IP packet will register with the INPUT chain when the packet is destined for the local system. The INPUT chain has further the MANGLE and then FILTER table. Firstly, the packet goes through MANGLE table, where, TOS (Type of services), DSCP, ECN, MARK, and ROUTE fields are modified as per the requirements. Secondly, the packet travels to the FILTER table where filtering of the IP packet is carried out for all incoming packets. Finally, after being passed from all chains and tables, the packet will be delivered to the appropriate process. Figure 9, details the overall packet flow, destined for the local system only.

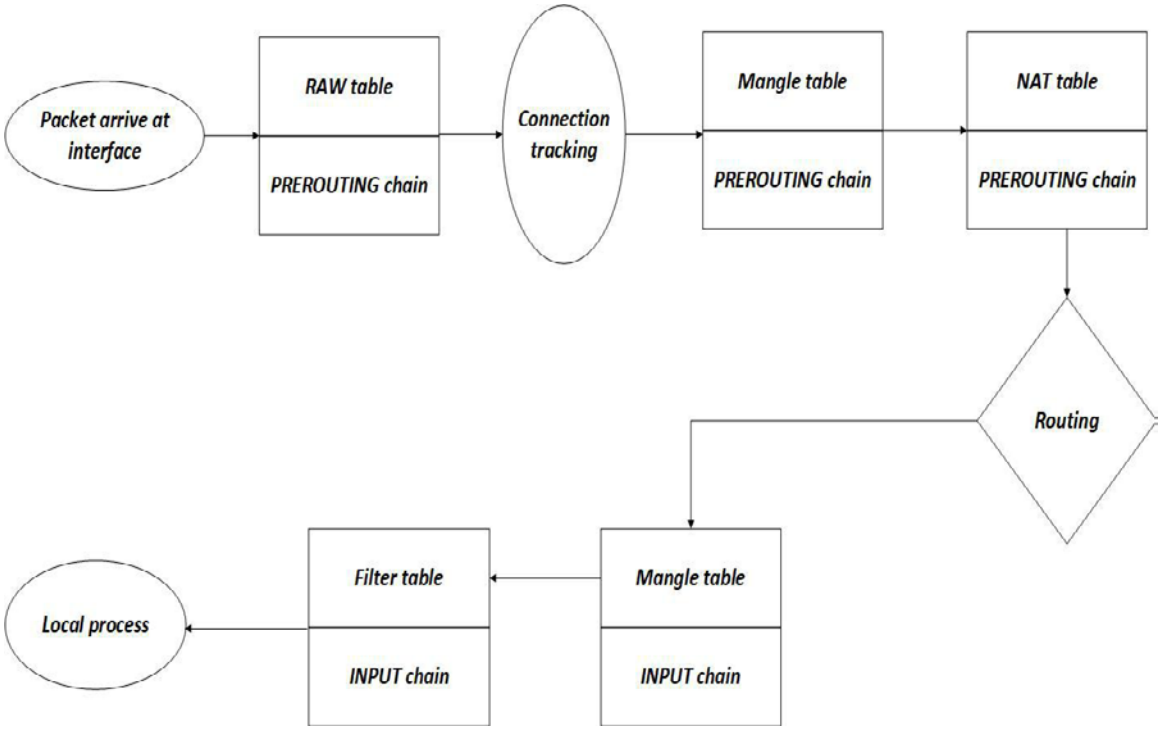


Figure 9: Packets Destined for Local system

– IP packets destined for the remote system

Similarly, when the packet is destined for a remote system, then the packet will hit the FORWARD chain. The FORWARD chain also consists of the MANGLE and the FILTER table, where respective packet fields are modified in MANGLE table, and further filtering of the packet is carried out. Moreover, now the IP packet travels through the POSTROUTING chain, where the packet again goes through MANGLE table and then to NAT table. In the NAT table MASQUERADE, source NAT (SNAT), and NETMAP are carried out. Finally, after being passed from all chains and tables, the packet will exit the output interface. Figure 10, details overall packet flow, destined for a remote system only.

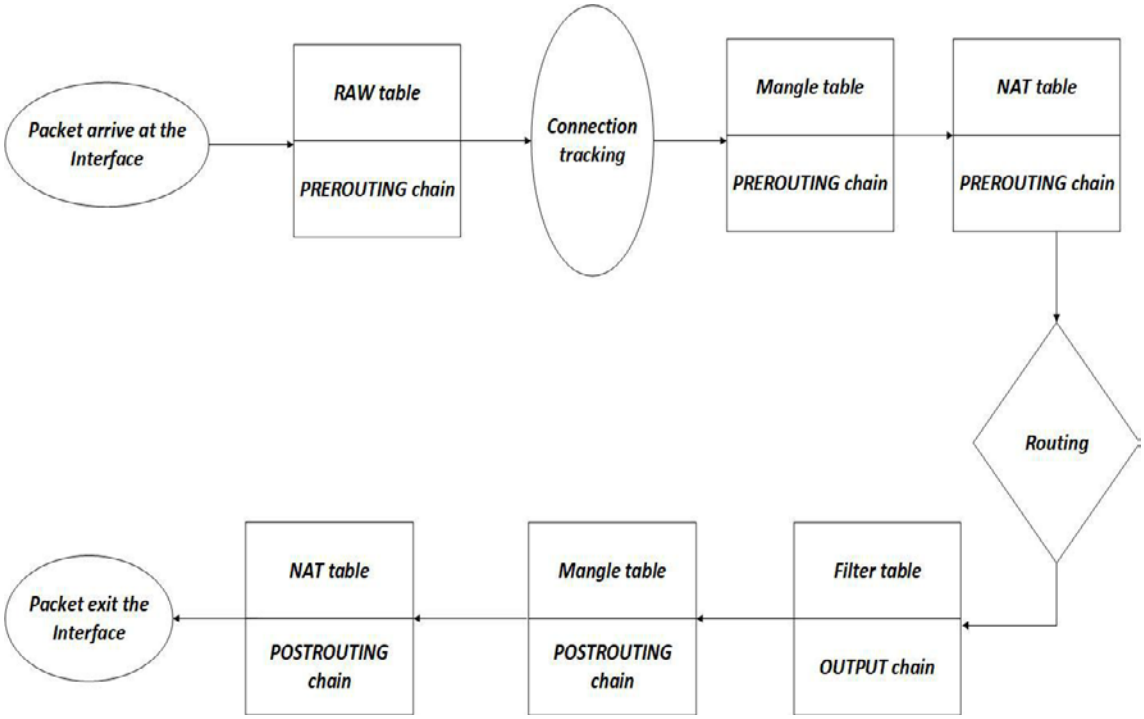


Figure 10: Packets Destined for remote system

– Locally generated IP packets

In addition, when a local process generates an IP packet, the packet will travel through the RAW, the MANGLE and the NAT table in the OUTPUT chain, where similar functionality is carried out as in the PREROUTING chain. For instance, adding the connection state entry to the connection tracking, mangling the packet fields and replacing the destination address (DNAT). Likewise, after routing table, the packet reached the FILTER table in the OUTPUT chain where filtering is carried out. Furthermore, the packet traverses through the POSTROUTING chain, where the packet again goes through the MANGLE table and then to the NAT table where MASQUERADE, SNAT, and NETMAP are carried out. Finally, after being passed from all the chains and table packets will exit the output interface. Figure 11, details overall packet flow, for locally generated packets only.

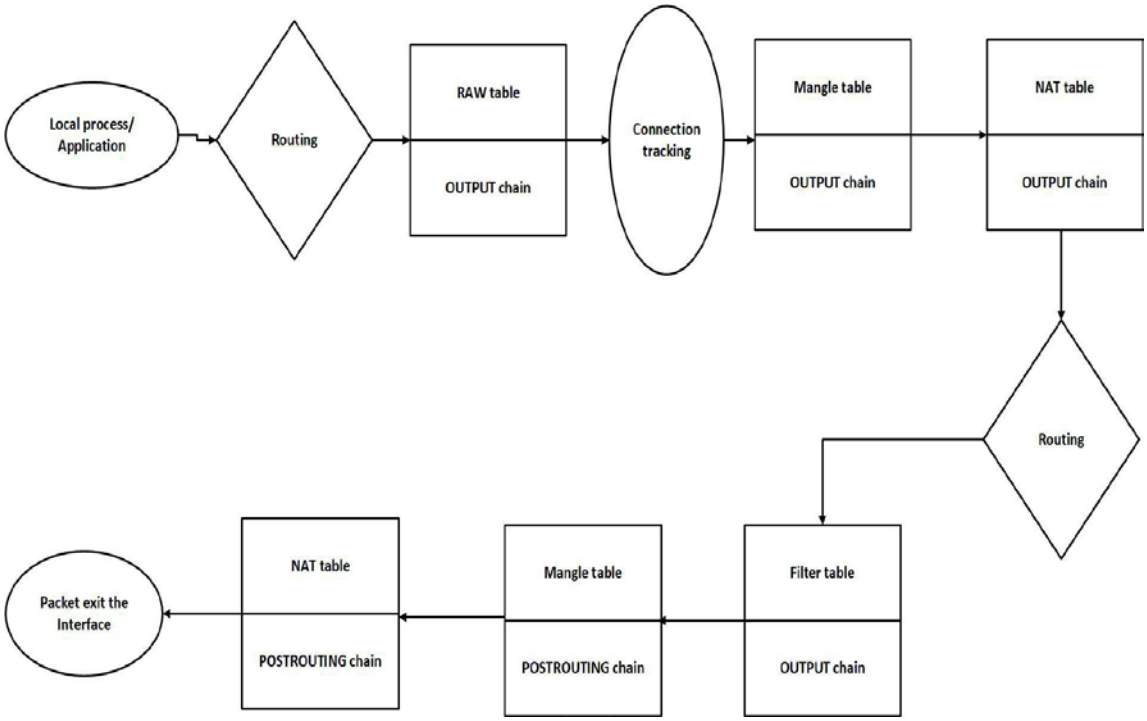


Figure 11: Packet generated by local system

2.4.3 IPTABLES

IPTABLES is the packet selection mechanism at user-space implemented by the Netfilter framework, which provides a transparent means for packet filtering and mangling. All the IP packets that enter the networking stack have to traverse through the different tables and chains. In these table and chain packets are filtered and mangled through the different rule sets. To, determine what types of packets will be dropped and which kinds of packets will be processed through iptables is based on the IP packet header information. [14]

Furthermore, iptables has been organized into different rule sets by ordering the different chains and tables. Netfilter chains are the sequential order of processing different rules and making decisions based on the different criteria found in the packet header. Netfilter has five different chains PREROUTING, INPUT, FORWARD, OUTPUT, and POSTROUTING. These chains are classified based on the type of action that needs to be performed during the packet processing. The tables are a functional group of packet processing, for example, packet filtering, address translation or packet mangling. The rules determine either the packet is forwarded to the upper layer for processing or simply dropped.

In addition, iptables operates at the transport and the network layer of the OSI stack for processing the IP packets and delivering to the appropriate destination of the packet. When the kernel receives the packets, IP layer processes the packet and compares the

and compares the packet information with the first rule in the rule sets. After reviewing the rule, the packet will be passed to a local process or sent to another host depending upon the property of the IP packet. If the packet did not match the first rule, then the next rule will be evaluated and so on until a matching rule is found. If no matching rule is found the packet will be discarded, which is defined in the default table-chain POLICY. Table 5 illustrates the different tables that are available in corresponding chains in Netfilter.

Table 5: Netfilter iptables and chains

Table / Chains	**PREROUTING**	**INPUT**	**FORWARD**	**OUTPUT**	**POSTROUTING**
RAW	*			*	
MANGLE	*	*	*	*	*
NAT	*	*		*	*
FILTER		*	*	*	
SECURITY		*	*	*	

2.5 Connection Tracking

Connection tracking is a block of Netfilter framework that is responsible for storing the information of all the network connections traversing through the network stack. Connection tracking provides a mechanism for the kernel to act as a stateful packet filtering firewall by keeping track of all the network connection states. Further-more, a particular framework called conntrack module in the kernel is responsible for handling all connection tracking. This module enables a stateful packet inspection for iptables as well for the NAT routing. Moreover, connection tracking manage all the active sessions traversing through the system and the state information is revoked/removed when a session is closed or the timeout value expires. [14] [17]

Furthermore, the connection tracking is handled in the RAW table of the PREROUTING chain for both the incoming and outgoing packets. For locally generated IP packets it is handled by the OUTPUT chain.

The connection entry is stored as a five-tuple element, i.e., protocol, source IP, source port, destination IP, and destination port for each unique entry. Figure 12, is an example of a connection tracking entry, the connection entry is for TCP protocol indicated by TCP keyword at the beginning of the connection state and by the protocol number as 6. This connection state will expire in 299 seconds, and the communication is in the ESTABLISHED state and will not be deleted from the table even if the table becomes full, which is indicated by the ASSURED keyword.

The conntrack entry is stored into two separate nodes (one for each direction) in different

```
tcp 6 299 ESTABLISHED src=10.128.254.24 dst=10.128.254.5 sport=22
dport=38210 src=10.128.254.5 dst=10.128.254.24 sport=38210
dport=22 [ASSURED] mark=0 use=1
```

Figure 12: Connection tracking entry

linked lists as shown in Figure 13. The hash value is calculated based on the received packet and used as an index in the hash table. Therefore, the connection states hash value will be stored in the two nodes of a bucket for both directions, i.e., the incoming and the outgoing IP packets. In addition, the linked list is known as buckets which are an element in a hash table. [18]

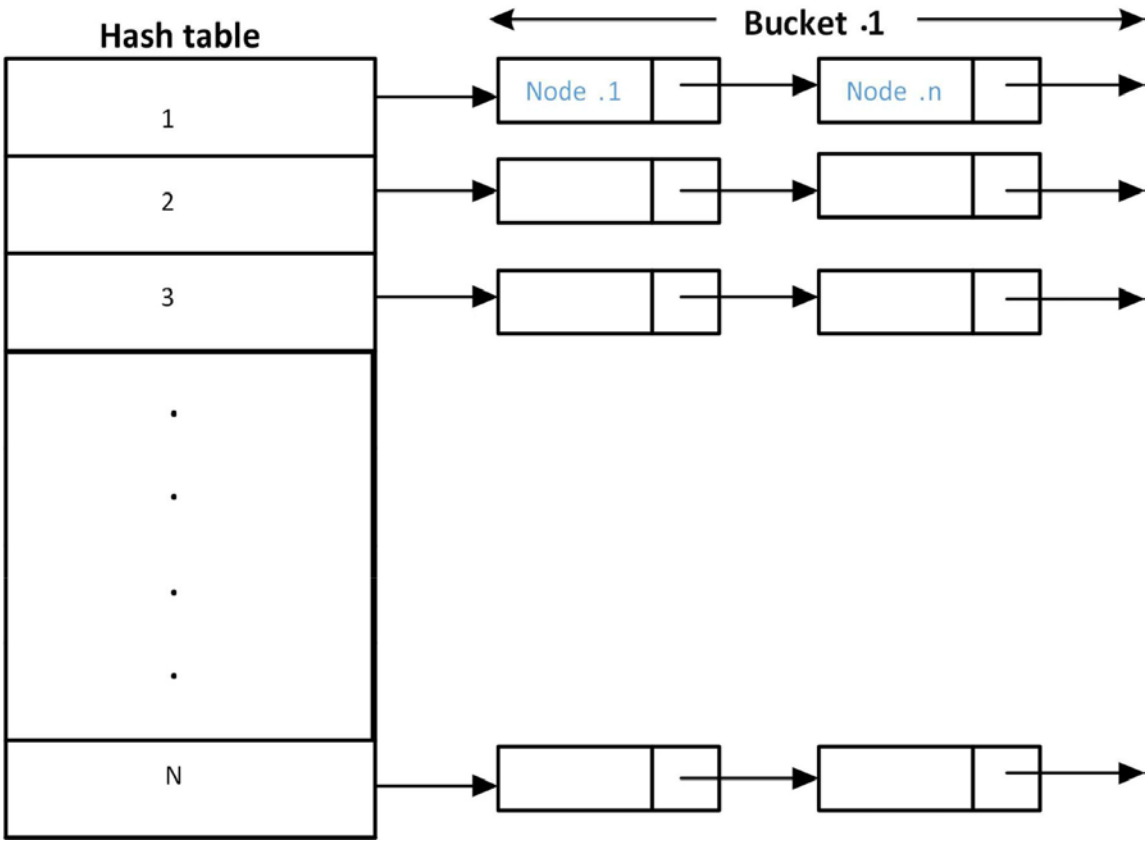


Figure 13: Hash Table and linked list

Furthermore, iteration is done over the linked list (length of a bucket) of the nodes to find an appropriate entry. The cost to find the distinct conntrack node depends on the length of the list and the position of the node in the list. Therefore, increasing the number of buckets decreases the number of nodes in the linked list and thus reduces the cost of iteration over the linked list. However, with a large number of buckets, there is a higher probability of buckets being empty and each bucket takes non-swappable physical memory. [18]

Netfilter defines two essential parameters in Linux kernel for handling the connection tracking namely, **nf_conntrack_max** and **nf_conntrack_buckets**. The **nf_conntrack_max** is used for handling the maximum number of connection entries, while the **nf_conntrack_buckets** define a maximum length of the hash-table or the maximum number of buckets. These two parameters are tunable allowing the maximum entries to be stored in the connection tracking. Furthermore, each connection entry takes a certain amount of non-swappable memory, and increasing the maximum limit consumes more memory resources.

In addition, both **nf_conntrack_max** and **nf_conntrack_buckets** are computed automatically according to the amount of available RAM. The maximum connections is equal to size of hash table multiplied by **Hash Table (HT)** load factor which is a command standard. The HT load factor is an average length of linked-list per bucket in hash table. For optimal performance maximum connections is equal to eight times the bucket- size, which is a command standard in Ubuntu systems. In addition, in a standard Linux system, which has 1GB RAM, the maximum number of connection is configured as 65536, whereas bucket size is set to 8192 (**nf_conntrack_max/8**). This means the system can store only 65536 number of entries in a table after that kernel starts to drop packets since table if full and it cannot keep track of more connections. [18]

2.5.1 Connection states

The connection tracking is further classified into four valid states for determining the connection status for a particular connection flow. These four states are NEW, ESTABLISHED, RELATED and INVALID.

Table 6: Connection states

State	Details
NEW	When the connection tracking detects an IP packets that is not associated with the existing connections states. The connection tracking puts that IP packets flow into the NEW state. The NEW state implies the first IP packet in the communications has been noticed. The typical example in connection tracking is of TCP communications, as soon as the conntrack sees the TCP SYN packet, conntrack will put that packet flow into NEW state. In the case of other protocols, the first packet might be different than the SYN packet; still conntrack will consider the NEW state.
ESTABLISHED	The state is changed from NEW to ESTABLISHED when conntrack has seen the traffic in both directions. For example, during TCP connection when the conntrack identifies the SYN/ACK packet, the state will be updated from NEW to ESTABLISHED. While a valid response has to be received from the remote end in the case of UDP protocol, and for the ICMP protocol, ICMP echo reply has to be seen.
RELATED	In the connection tracking, any IP flow that is not a part of an existing connection, but related to another ESTABLISHED connection state, then the IP flow will be marked as RELATED. For example, FTP-connection is considered as RELATED. INVALID & The IP packets that are not identified to be part of any connection states are considered as INVALID.
INVALID	The IP packets that are not identified to be part of any connection states are considered as INVALID.
UNTRACKED	The packets that are marked in the RAW tables as NOTRACK is considered as the UNTRACKED state and these states are not stored in a conntrack table.

2.5.2 TCP state

TCP communication has three stages called three-way handshake as illustrated in Figure 14.

The first packet in the TCP session is always the SYN packet sent to the destination host and the destination host replies with the SYN/ACK packet. As soon as the conntrack detects the SYN packet a NEW state is created with keyword SYN_SENT, meaning only one direction traffic has been tracked. When replay traffic from the remote end is seen, i.e., SYN/ACK packet, the state is updated to SYN_RECV, implying that traffic in both directions has been identified. The SYN_RECV is an intermediate state between NEW and ESTABLISHED, immediately upon receiving the SYN/ACK packet the state is updated to ESTABLISHED as shown in Figure 15.

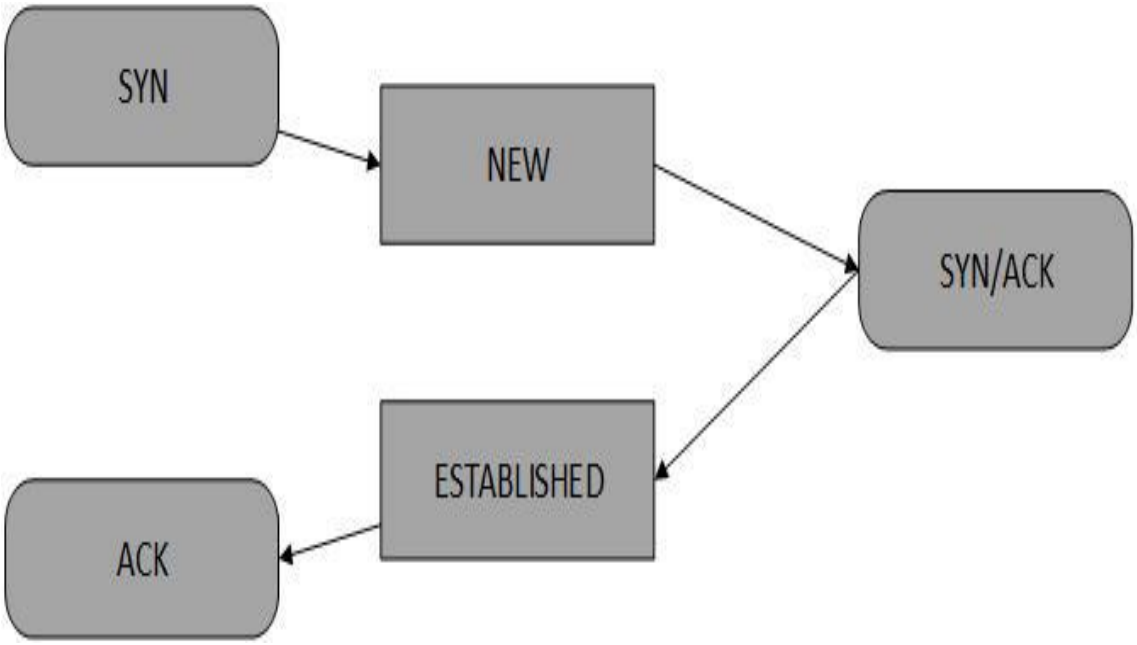


Figure 14: TCP connections establishing

During the ESTABLISHED state, actual data packets are transferred between the hosts and the state is put into the ASSURED mode. The ASSURED mode implies that the connection state information will not be removed from the conntrack table if the table becomes full. [10]

At some point in time, the connection state will change from ESTABLISHED to TIME_WAIT state before terminating the session. The TIME_WAIT state is considered as a buffer time for any lost or delayed packets received after a connection is closed and to avoid misunderstanding at the transport layer. [17]

The TCP connection has predefined timeout values for each connections state and Table 7 shows the default timeout values for different TCP states.

```
[NEW] tcp 6 120 SYN_SENT src=192.168.1.4 dst=109.105.109.249 sport=62483
dport=80 [UNREPLIED] src=109.105.109.249 dst=192.168.1.4 sport=80
dport=62483
[UPDATE] tcp 6 60 SYN_RECVsrc=192.168.1.4 dst=109.105.109.249 sport=62483
dport=80 src=109.105.109.249 dst=192.168.1.4 sport=80 dport=62483
[UPDATE] tcp 6 432000 ESTABLISHED src=192.168.1.4 dst=109.105.109.249
sport=62483 dport=80 src=109.105.109.249 dst=192.168.1.4 sport=80
dport=62483 [ASSURED]
[UPDATE] tcp 6 120 FIN_WAIT src=192.168.1.4 dst=109.105.109.249
sport=62483 dport=80 src=109.105.109.249 dst=192.168.1.4 sport=80
dport=62483 [ASSURED]
[UPDATE] tcp 6 30 LAST_ACK src=192.168.1.4 dst=109.105.109.249
sport=62483 dport=80 src=109.105.109.249 dst=192.168.1.4 sport=80
dport=62483 [ASSURED]
[UPDATE] tcp 6 120 TIME_WAIT src=192.168.1.4 dst=109.105.109.249
sport=62483 dport=80 src=109.105.109.249 dst=192.168.1.4 sport=80
dport=62483 [ASSURED]
```

Figure 15: TCP connection entry

Table 7: TCP state timeout

State	Timeout value
NONE	30 minutes
ESTABLISHED	432000 seconds
SYN_SENT (NEW)	120 seconds
SYN_RECV	60 seconds
FIN-WAIT	120 seconds
TIME_WAIT	120 seconds
CLOSE	10 seconds
CLOSE-WAIT	12 hours
LAST_ACK	30 seconds
LISTEN	120 seconds

2.5.3 UDP state

UDP is a connection-less protocol, that is UDP does not try to establish a session first, as TCP does before actual communication could begin. Therefore, when the connection tracking sees the first UDP packet conntrack will create a NEW state

with the keyword UNREPLIED as shown in Figure 17. When the reply traffic from destination end is observed, then the NEW state is updated to ESTABLISHED and the state is moved to ASSURED mode. Figure 16 depicts UDP communication with appropriate state. [17]

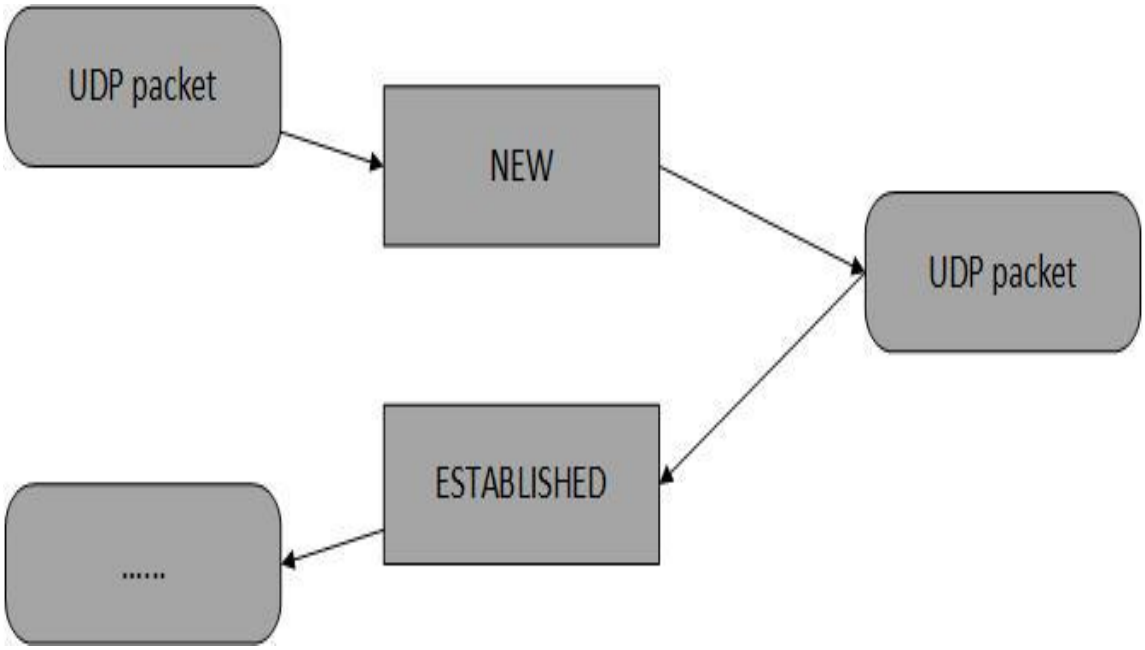


Figure 16: UDP connections establishing

```
[NEW] udp 17 30 src=192.168.1.4 dst=130.233.251.4 sport=37233
dport=53 [UNREPLIED] src=130.233.251.4 dst=192.168.1.4 sport=53
dport=37233
[UPDATE] udp 17 180 src=192.168.1.4 dst=130.233.251.4 sport=37233
dport=53 src=130.233.251 dst=192.168.1.4 sport=53 dport=37233
[ASSURED]
```

Figure 17: UDP connection entry

Similar to TCP, UDP also has predefined timeout values. However, UDP has only two timeout values, and they are relatively small and mostly suitable for all UDP applications.

Table 8: UDP state Timeout

State	Timeout value
NEW	30 seconds
ESTABLISHED	180 seconds

2.5.4 ICMP state

ICMP protocol will not establish any session during the ICMP communication. However, ICMP packets are also stored in conntrack as valid states just like TCP or UDP. All ICMP messages will have two states NEW and ESTABLISHED, and Figure 18 illustrates the ICMP connections state in connection tracking system.

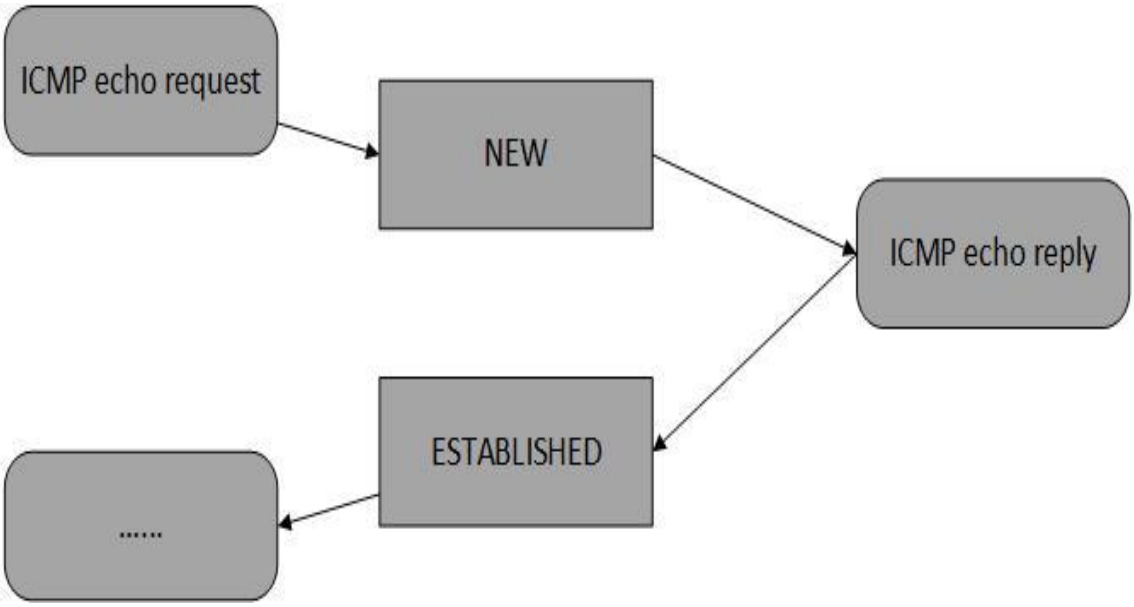


Figure 18: ICMP connections state

The first packet in ICMP communication is the ICMP echo request message, which is considered as a NEW state in conntrack. And upon receiving ICMP echo reply message from the remote end, the state is updated to ESTABLISHED. In case of ICMP protocol, ESTABLISHED state is not marked as ASSURED because ICMP doesn't establish a session. Figure 19 shows the ICMP connection entry in connection tracking.

Furthermore, if an ICMP packet is lost or denied by the remote host or the network is unreachable, then, the initial state will be NEW in connection tracking. However, the ICMP error message as network unreachable is recognized as the RELATED state, and ICMP error massage will be delivered to originating node. [17]

```
[NEW] icmp     1 30 src=192.168.1.4 dst=172.217.22.174 type=8 code=0
id=5161 [UNREPLIED] src=172.217.22.174 dst=192.168.1.4 type=0 code=0
id=5161
[UPDATE] icmp     1 29 src=192.168.1.4 dst=172.217.22.174 type=8 code=0
id=5161 src=172.217.22.174 dst=192.168.1.4 type=0 code=0 id=5161
```

Figure 19: ICMP connections entry

ICMP protocol has only one timeout value.

Table 9: ICMP state Timeout

State	Timeout value
ICMP	30 seconds

2.5.5 Default connection

When the connection tracking system is unaware of any specific protocols, and then conntrack will handle these protocols state as the default connection. These connections are treated similarly to UDP connections. However, timeout values for these states are different. The default timeout value is 600 seconds. [10]

2.5.6 Untracked connections

The packets that are marked in the RAW tables as NOTRACK are considered as UNTRACKED and are not stored in the conntrack table. [10]

2.5.7 Connection states timeout values on physical devices

Below are some key default timeout values for different protocols.

```
sysctl -a | grep nf | grep timeout
net.netfilter.nf_conntrack_generic_timeout = 600
net.netfilter.nf_conntrack_icmp_timeout = 30
```

net.netfilter.nf_conntrack_icmpv6_timeout = 30

net.netfilter.nf_conntrack_tcp_timeout_close = 10

net.netfilter.nf_conntrack_tcp_timeout_close_wait = 60

net.netfilter.nf_conntrack_tcp_timeout_established = 432000

net.netfilter.nf_conntrack_tcp_timeout_fin_wait = 120

net.netfilter.nf_conntrack_tcp_timeout_last_ack = 30

net.netfilter.nf_conntrack_tcp_timeout_max_retrans = 300

net.netfilter.nf_conntrack_tcp_timeout_syn_recv = 60

net.netfilter.nf_conntrack_tcp_timeout_syn_sent = 120

net.netfilter.nf_conntrack_tcp_timeout_time_wait = 120

net.netfilter.nf_conntrack_tcp_timeout_unacknowledged = 300

net.netfilter.nf_conntrack_udp_timeout = 30

net.netfilter.nf_conntrack_udp_timeout_stream = 180

3 Testbed setup

This chapter gives an overview of the test setup for performance testing and all the different tools, applications, and scripts that are used to carry out the performance testing of PRGW.

3.1 Environment setup

The entire test for connection tracking, hash-table performance, and CPU performance has been evaluated under the single test environment which has been virtualized on the Linux machine. The physical details of the testing Linux machine are shown Table 10.

Table 10: Hardware details of Physical server

Architecture	x86_64
Kernel version	4.8.0-54-generic
Physical memory	32G
CPU op-mode(s)	32-bit, 64-bit
CPU(s)	24
Thread(s) per core	2
Core(s) per socket	6
NUMA node(s)	2
Vendor ID	6
CPU family	Genuine Intel
Model	45
Model name	Intel(R) Xeon(R) CPU E5-2630 0 @ 2.30GHz
CPU MHz	1244.000
CPU max MHz	2300.0000
CPU min MHz	1200.0000

Furthermore, the test environment has been configured by using the virtual Ethernet interfaces, which are always available in pairs. Two pairs of virtual Ethernets namely source0s and source0 as well as sink0s and sink0, have been created. Furthermore, the ARP broadcast has been disabled, to use the interface with its own MAC address as source and destination of Ethernet frames while sending the packets. In addition, interface source0 has been configured with IP address 10.0.0.1/8, making possible to use all available IP addresses of the class via source0s. Like-wise, sink0 has been configured with 100.64.0.1/8 IP addresses making possible to use the entire 100th range for addressing via sink0s. Therefore, using such a large address space in two networks allows 2^{24} 2^{24} numbers of host-to-host IP communications. Figure 20 represents the test setup.

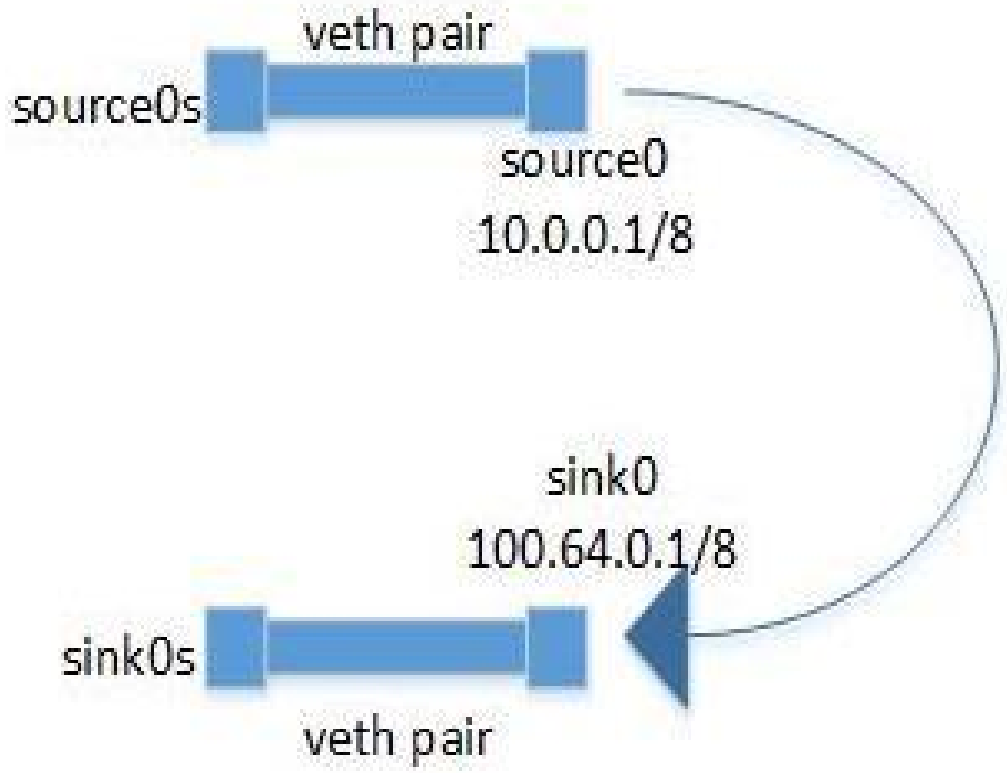


Figure 20: Test Setup for Connection tracking and IP tables

Moreover, Linux by default does not route any IP packets. In order to make Linux machine to route the IP packets and act as a router, ***ip -forwarding*** has to be enabled in the Kernel. The IP forwarding mechanism has been configured by adding the command net.ipv4.ip_forward=1. The configuration setup has been made persistent even after boot, by editing the /etc/sysctl.conf file to add IP forwarding mechanism.

3.2 Test tools

To be able to carry out the testing processes, different types of network tools and applications have been used. For example, Ostinato to generate test traffic, tcpdump for capturing packets, tcprewrite for mangling with packets fields. And finally, tcpreplay for sending the IP traffic across the network. In addition, UDP protocol has been selected for testing because, UDP is a light-weight protocol, it is very fast, it has less overhead and finally, give more performance than connection oriented protocol.

3.2.1 Tcpdump

Tcpdump is a network traffic capturing tool widely used in the Linux system. It is used to capture or filter network traffic traversing through the network devices for analyzing the traffic. Further, tcpdump can be used to dump the IP packets capture to the console or writing to files.[20] During the testing phase, network traffic has been captured in different files. Later a capture file has been used to generate network traffic for performance testing. Following is the tcpdump command used to capture the traffic on interface source0 and writing capture to the file 1M_source0.2.pcap.

```
tcpdump -nli source0 -w 1M_source0.2.pcap –B 65536
```

Moreover, the mergecap command has been used to merge two or more pcap files to make single pcap file comprising a large number of packets.

```
mergecap 1M_source0.2.pcap 1M-source0.3.pcap –w merged_file.pcap
```

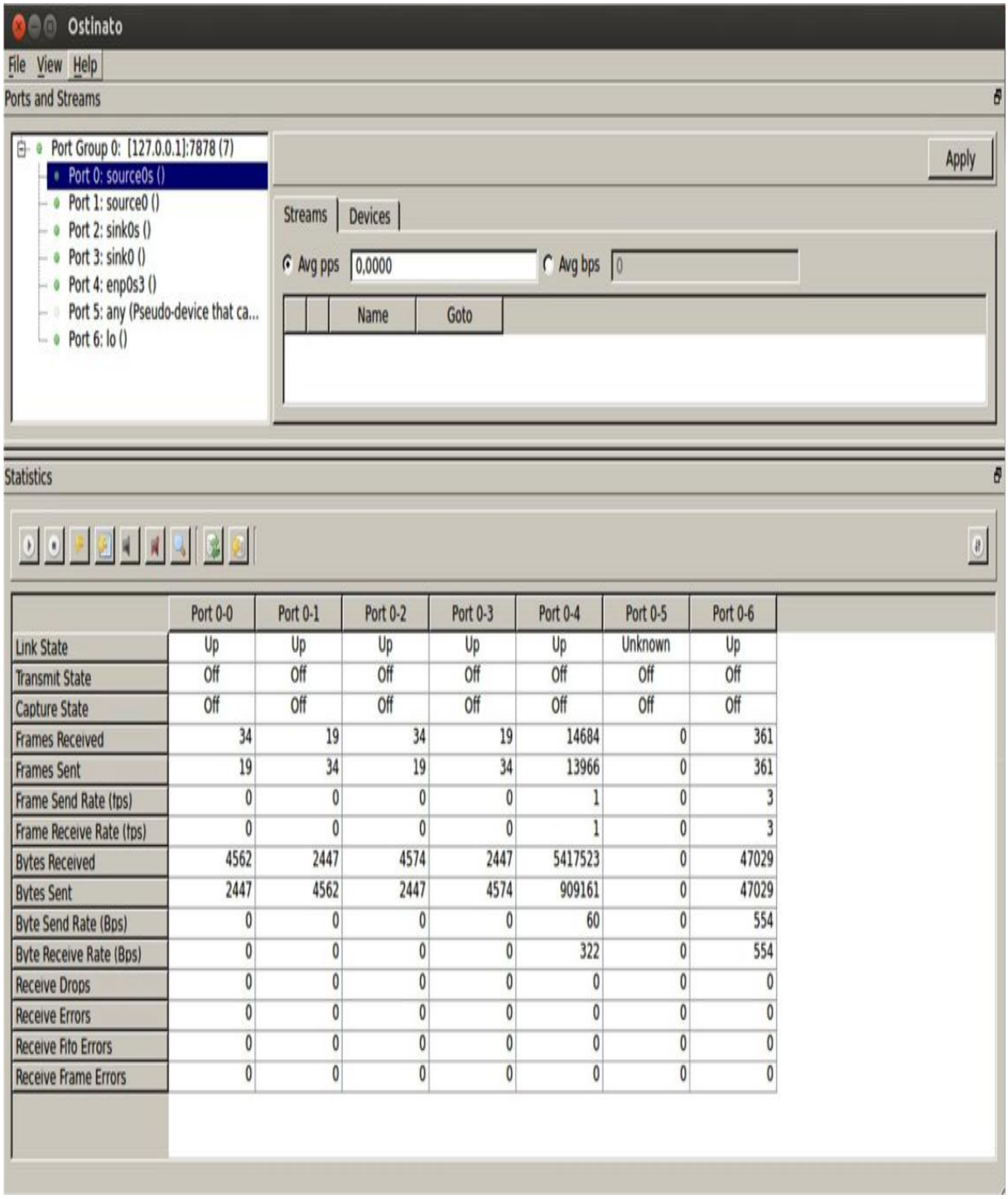
The above command was used to combine two pcap files containing 1M packets to create single pcap file with 2M packets.

3.2.2 Ostinato

Ostinato is an open-source and GUI network traffic generator for Windows and Linux system. It offers a robust python API for network test automation and load testing. Ostinato is composed of two components, the controller, and the drone. The controller could be a simple GUI running on a client machine or the client running a simple python script. The drone, however, is a primary component of Ostinato responsible for creating, sending and capturing traffic whereas, in the default mode of Ostinato, the controller together with the drone is merged into a single architecture. [19] And, this single architecture of Ostinato was used for generating the network traffic during the testing of PRGW.

The Ostinato is capable of generating a high volume of network traffic approximate to the transmission speed of Ethernet adapter. In addition, it can create all kinds of packets with different protocols including the higher layer payload in packets as well. Ostinato was used to generate sample packets by randomizing IP addresses only, without having to worry about the upper layer protocols. Moreover, Ostinato can be used to send, receive and capture the traffic across the network at different rates.

Figure 21 shows the GUI layout of Ostinato. Ostinato uses a port group for all the available interfaces that are controlled by the Ostinato. These port groups can be created, deleted and modified, including the remote network devices that can be added to the port group. In addition, particular interface have to be selected to be able to generate network traffic. [19]



	Port 0-0	Port 0-1	Port 0-2	Port 0-3	Port 0-4	Port 0-5	Port 0-6
Link State	Up	Up	Up	Up	Up	Unknown	Up
Transmit State	Off	Off	Off	Off	Off	Off	Off
Capture State	Off	Off	Off	Off	Off	Off	Off
Frames Received	34	19	34	19	14684	0	361
Frames Sent	19	34	19	34	13966	0	361
Frame Send Rate (fps)	0	0	0	0	1	0	3
Frame Receive Rate (fps)	0	0	0	0	1	0	3
Bytes Received	4562	2447	4574	2447	5417523	0	47029
Bytes Sent	2447	4562	2447	4574	909161	0	47029
Byte Send Rate (Bps)	0	0	0	0	60	0	554
Byte Receive Rate (Bps)	0	0	0	0	322	0	554
Receive Drops	0	0	0	0	0	0	0
Receive Errors	0	0	0	0	0	0	0
Receive Fifo Errors	0	0	0	0	0	0	0
Receive Frame Errors	0	0	0	0	0	0	0

Figure 21: Ostinato GUI interface

Figure 22 shows the edit stream tab, where the IP packets are created. This section is used to select different data-link protocols like Ethernet II, 802.3Raw and, many more In addition, the network layer protocols, for example, ARP, IPv4 or IPv6 are also available Furthermore, not only TCP or UDP but also ICMP, and multicast messages are available as transport layer protocols. However, during this test, UDP has been the primary focus. Finally, the upper layer information could be selected as per the requirements of the packets.

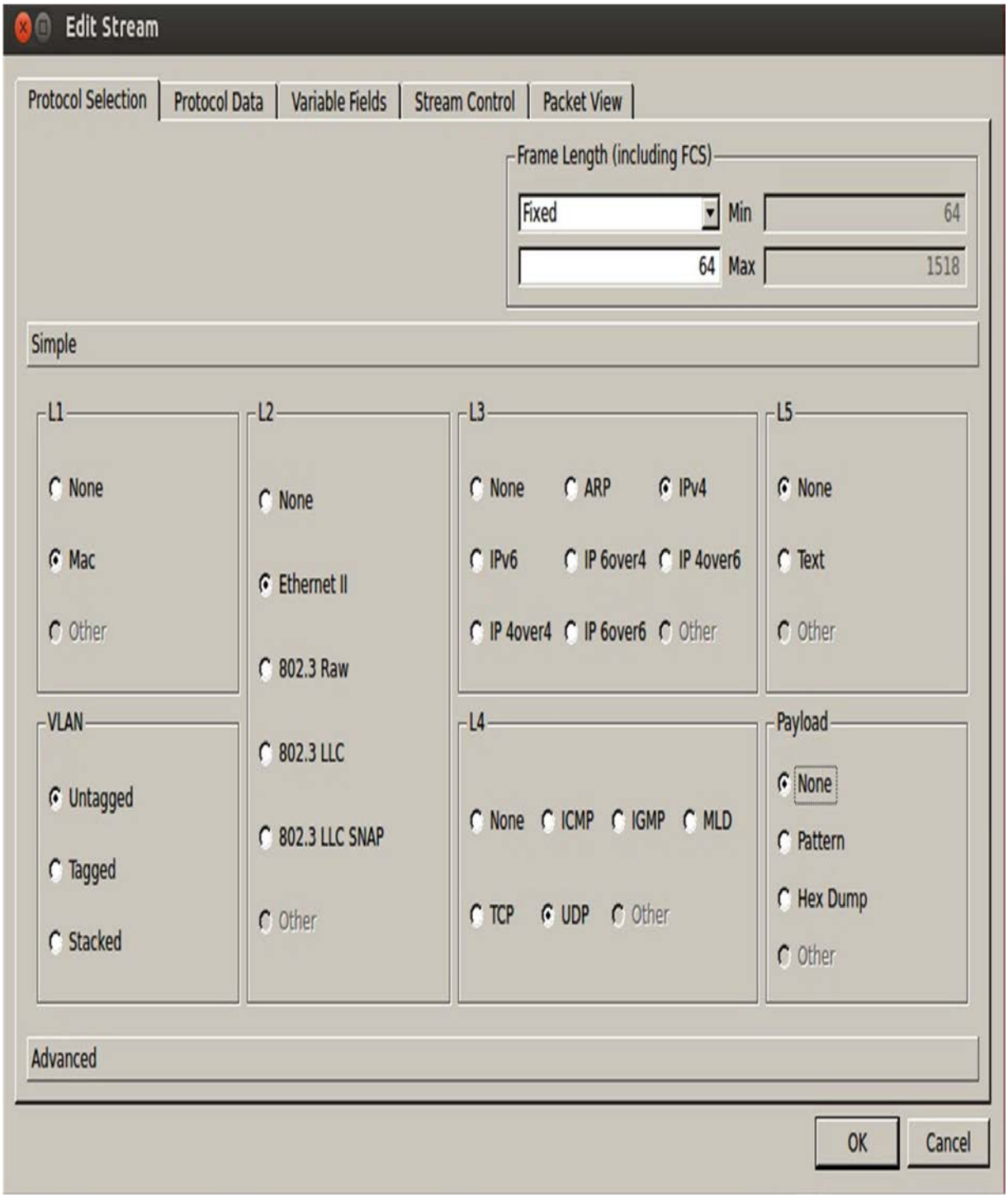


Figure 22: Ostinato crafting packet

Moreover, the UDP packets have been generated up to the transport layer for testing of PRWG. Also, the random source and destination UDP port have been configured, allowing more randomness in the IP packets. Finally, one million sample IP packets were created. Figure 23 shows simple packet structure designed by the Ostinato for testing purpose. Further, Table 11 illustrates the properties of the packet.

Table 11: Test packet properties

Field	Value
Frame length	64 bytes
Source MAC	00:00:00:00:01:02
Destination MAC	00:00:00:00:01:01
Source IP	10.0.0.2-10.255.255.254
Destination IP	100.64.0.2-100.255.255.254
Source port	1-65535
Destination port	1-65535
UDP checksum	0x0000

Finally, using Ostinato 1 million UDP sample packets were created, these packets were random IP source and destination along with random source and destination ports numbers. However, the randomness causes Ostinato to generate IP packets with a bad checksum. The IP packets with bad checksum would not create a connection state in connection tracking. The kernel will try to verify the checksum and the packets with bad checksum will be dropped. Further, the bad checksum was a bug in Ostinato, therefore, in order to overcome the problem; tcprewrite have been used to fix the bad checksum.

```
tcprewrite --fixcsum --infile=64k_sample.pcap --outfile=64k_fix_csum.pcap
```

Above command was used to fix the bad checksum from sample capture file, –fixcsum is the argument which rewrites the checksum from the input file (–infile) and writes to a new file (–outfile)

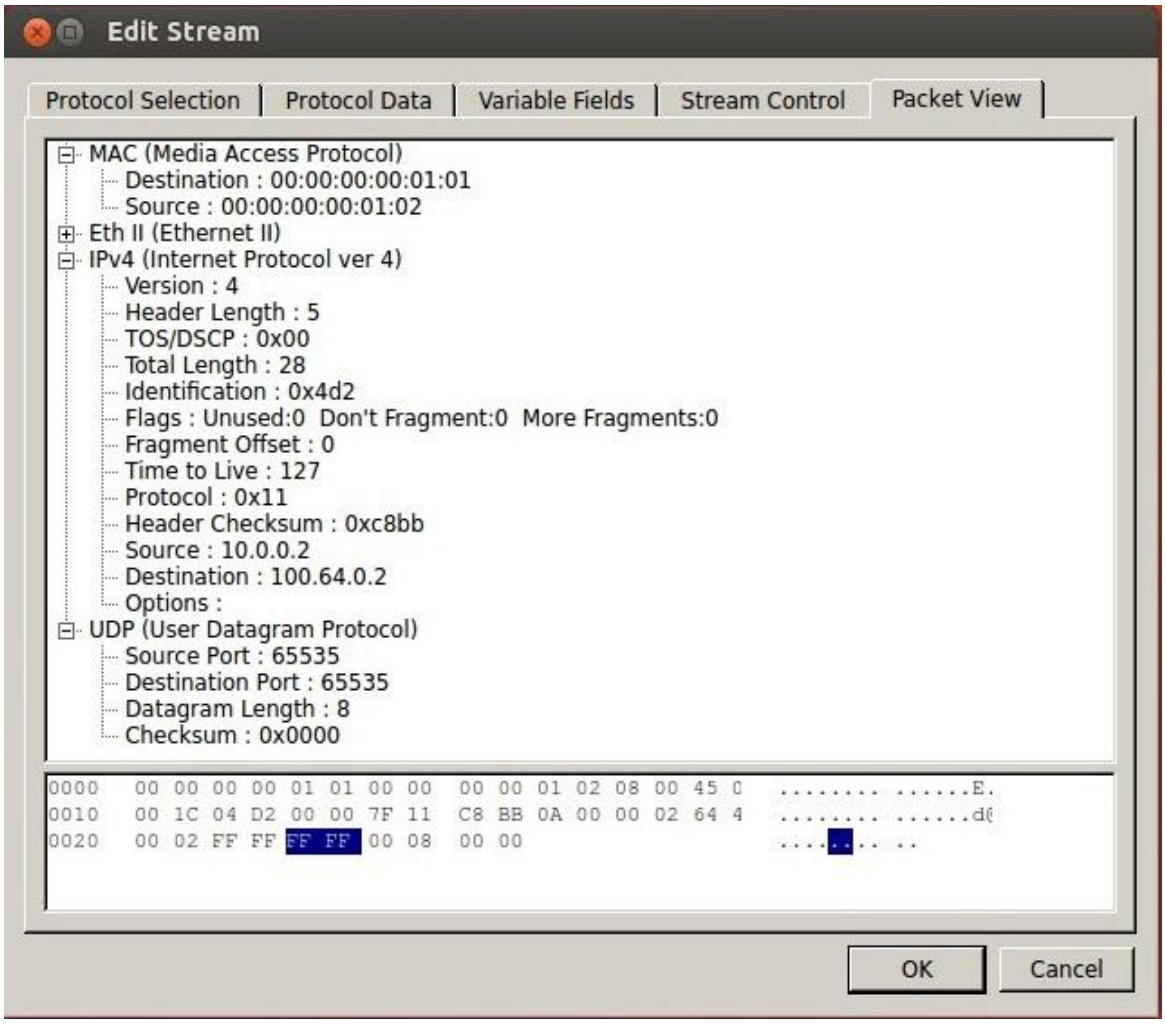


Figure 23: Packet View in Ostinato

3.2.3 Tcprewrite

Tcprewrite is used to rewrite different IP packet fields, for instance, Layer 2 & Layer 3 addresses, bad checksum, TCP/UDP port numbers from pcap files. Furthermore, tcprewrite can be used to create various pcap files with unique source IP and destination IP. This tool is used because of the problem caused by ostinato during packet generations. Tcprewrite is used primarily to fix IP checksum and to create a large number of IP packets with unique source and destinations. [23].

Using this tool 16M different IP packets were created. The following is an ex-ample of rewrite source and destination IP of sample 1M packets.

tcprewrite –pnat=10.0.0.0/16:10.1.0.0/16,--pnat=100.64.0.0/16:100. 65.0.0/16 --infile = 1M_fixed_csum.pcap --outfile=1M_subnet_41.pcap

In the above command sample, the pcap (1M_fixed_csum.pcap) file has been rewritten with a different source, and destination IP addresses to a new pcap file: 1M_subnet_1.pcap. The sample file contains packets with the source from **10.0.0.0/16** network that has been changed to the **10.1.0.0/16** subnet. And the destination address from **100.64.0.0/16** network has been changed to **100.65.0.0/16** network by using the –**pnat** flag.

3.2.4 Tcpreplay

Tcpreplay is the open source network utility for editing and replaying the previously captured network traffic. It is a handy tool for testing the performance of different network devices, for instance, router, firewalls, switches, network intrusion prevention system and intrusion detection system (IPS and IDS). Tcpreplay provides detailed network analysis, and the statistics for the network performance. For in-stances, the total number of packets, throughput in terms of PPS, as well as flows per second (FPS), the total number of bytes sent, packet loss during the transmission. [21]

In addition to network statistics, tcpreplay can send the IP packets across the network at very high speed, using the previously captured pcap files. Further, large numbers of packets can be transmitted from a single or the multiple interface. In the default mode, tcpreplay can send network traffic at the same speed that tcpdump or Wireshark has captured the packets. However, the rate at which packets are transmitted can be manually configured. Tcpreplay provides a fully scalable and reputable means of replaying pcap files even at high-speed network such as the 10GigE network.

```
tcpreplay --topspeed --intf1=source0s 1M-packet_cthashing.pcap
```

The above command was used for replaying network traffic and following are the flags used:

- - **topspeed** is used to send packets as fast as possible

- - **intf1** is used to indicate interface that traffic is being addressed and

- - **1M_packet_cthashing.pcap** is the pcap file that tcpreplay is transmitting onto the network.

Other important flags/options:

- - **enable-file-cache** this option allows to cache a pcap file to physical memory

- - **loop=num** it loop pcap file for given amount of times.

- - **limit = num** manually sets the number of packets to be sent

- - **topspeed** sends the packet as fast as possible.

- - **intf1= string** interface from which traffic is sent

- - **pps-multi = num** number of packets to send for each time interval.

- - **preload-pcap** preloads packets into the RAM before sending.[22]

3.2.5 Scripts

A different script file has been created to ease out the tremendous amount of work for filtering the data and running various commands repeatedly. For each set of tests, separate script file has been made which has been included in the appendix sections.

4 Test and Evaluation

This chapter describes an overview and the purpose of each test along with a comprehensive description. Altogether, there are three testing categories in this thesis, which is further divided into the respective smaller test cases. And finally, the results have been gathered from the different test cases, their corresponding graphs and analysis have been discussed.

4.1 Connection tracking

First and foremost, the connection tracking test has been carried out for evaluating the performance of conntrack module to store the connection state of the IP packets, and analyzing the impact of the conntrack module on the Linux machine. Further, the test has been conducted using the UDP protocol which has the default timeout value of 30 seconds.

Command used to view the UDP timeout value:

```
sysct -a | grep conntrack_udp
net.netfilter.nf_conntrack_udp_timeout = 30
```

Because the default timeout value is very small for the UDP protocol and is not sufficient time for the testing purpose. The default timer has to be changed, to make sure that connection states will not expire while the test is running. For this reason, the UDP timeout value has been increased by 1000 times, i.e., (30000seconds), and the following command has been used to fix the timeout problem.

```
sysctl -w net.netfilter.nf_conntrack_udp_timeout=30000
```

Moreover, during the test of connection tracking, two primary variables in Netfilter system have been tuned namely, nf_conntrack_max and nf_conntrack_buckets. The nf_conntrack_max define the maximum number of connection entries that can be stored by the connection-tracking table. While the nf_conntrack_bucket, specify the size of the hash table for storing the list of connection entries.

Commands used:

```
sysct -a | grep conntrack_max

net.netfilter.nf_conntrack_max = 262144

sysct -a | grep conntrack_buckets

net.netfilter.nf_conntrack_buckets = 65536
```

In addition, these parameters will be tuned as per requirements of the test and its objective.

4.1.1 Scaling million connections

The primary objective is to test the feasibility of scaling the number of connections on connection tracking to millions.

Firstly, different values of the nf_conntrack_max and nf_conntrack_buckets were tuned to verify that Linux kernel would take the provided values or not. The following commands have been used to investigate the objectives of the test.

```
sysctl -w net.netfilter.nf_conntrack_max = 65536
sysctl -w net.netfilter.nf_conntrack_buckets = 65536
```

Furthermore, both the nf_conntrack_max and nf_conntrack_buckets values were kept identical throughout the test period. Since the primary goal of this test was to verify if Linux kernel would accept the provided values.

Table 12 illustrates the different values of the nf_conntrack_max and the nf_conntrack_buckets that were implemented on Linux kernel.

Further, it can be visualized that kernel took almost all the assigned values, without any errors. However, after the threshold value for both the parameters, kernel starts to reject the given values. The maximum number of connections sup-ported by the Linux kernel is 536,870,912 and for the nf_conntrack_buckets is 268,435,456. Therefore, beyond these limits, the system did not accept any values, instead, give an error as the kernel cannot allocate memory.

Hence, the maximum values that can be configured on Linux Kernel for both the nf_conntrack_max and the nf_conntrack_buckets have been determined to be 536,870,912 and 268,435,456 respectively. Moreover, as the size of the hash table, i.e., nf_conntrack_buckets start to increase, so does the physical memory, even though no connections have been offered.

Therefore, from the test, it could be concluded that as the size of buckets starts to increase, so is the physical memory used by buckets also increase. Furthermore, there is a hard-coded limit on both the number of maximum connections and the bucket size provided by the Linux kernel regardless of available physical memory.

Table 12: Netfilter values for nf_conntrack_max and nf_conntrack_buckets

nf conntrack max	nf conntrack buckets	Status
65,536	65,536	Kernel took value
131,072	131,072	Kernel took value
262,144	262,144	Kernel took value
524,288	524,288	Kernel took value
1,048,576	1,048,576	Kernel took value
2,097,152	2,097,152	Kernel took value
4,194,304	4,194,304	Kernel took value
8,388,608	8,388,608	Kernel took value
16,777,216	16,777,216	Kernel took value
33,554,432	33,554,432	Kernel took value
67,108,864	67,108,864	Kernel took value
134,217,728	134,217,728	Kernel took value
268,435,456	268,435,456	Kernel took value
536,870,912	536,870,912	Kernel throws an error as it cannot allocate memory for the nf_conntrack_buckets while accepting nf_conntrack_max value
1,073,741,824	1,073,741,824	Kernel throws and error as it cannot allocate memory for the nf_conntrack_buckets and invalid argument for nf_conntrack_max
2,147,483,648	2,147,483,648	Kernel throws an error as invalid argument for both the nf_conntrack_buckets and the nf_conntrack_max

4.1.2 Physical memory use by Connection tracking

The objective is to test the physical memory consumed by the connection states, and the number of connections that could be offered to connection tracking.

First and foremost, the test traffic has been sent from the interface sources0s destined to the interface sink0s as shown in Figure 24. The IP traffic is routed from source0 to sink0 and the connection state for IP packets will be created at the connection tracking. Therefore, sending the traffic with a different number of packets will create different number of states in connection tracking. And thus, the corresponding memory usage by connection tracking can be observed.

Furthermore, the nf_conntrack_max value has been changed accordingly to support the number of connection that has been offered. For example, for the 1 million number of connections offered, the nf_conntrack_max value has been set to 1048576. In addition to nf_conntrack_max, the nf_conntrack_buckets has also been changed with the HT factors (i.e. 1,2,4,8,16,32,64,128 and 256) to see the memory usages.

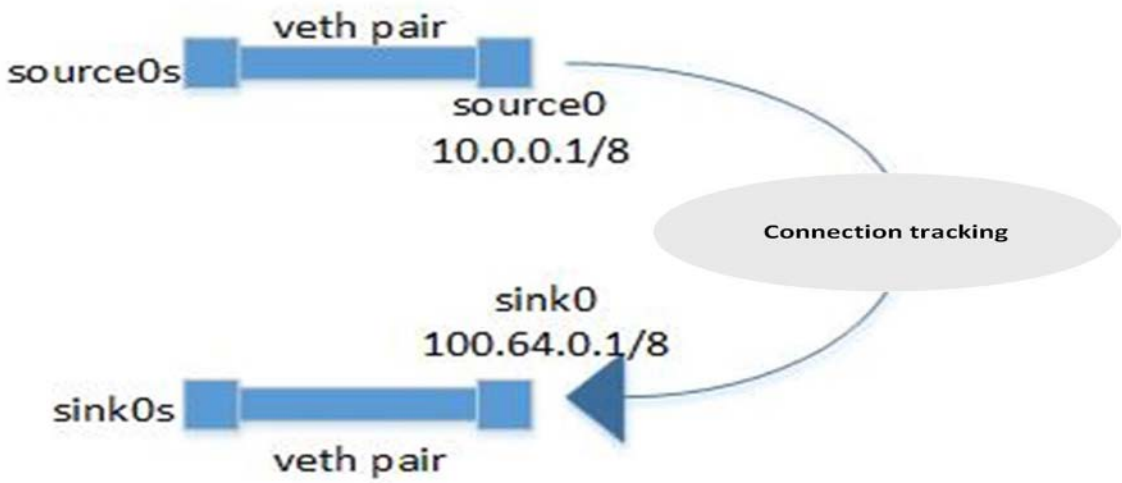


Figure 24: Test Setup for Connection tracking

sysctl -w net.netfilter.nf_conntrack_max = 1048576
sysctl -w net.netfilter.nf_conntrack_buckets = 65536

Table 13 details the different number of connection states that were created in connection tracking and the corresponding physical memory used by each number of connections. In the connection tracking, a single connection takes 320Bytes of memory, and 1M connections would take 320MB of physical memory as shown in Table 13. [18]

Table 13: Connection tracking memory utillization

Connection offered	Memory used(MB)
1 Millions	320
2 Millions	640
4 Millions	1280
8 Millions	2560
16 Millions	5120

In addition, Table 14 demonstrates the change of nf_conntrack_buckets sizes, for the 8M number of connections. In spite of changing the bucket size from small to the large value, the memory used by the connection tracking remains the same.

Table 14: Connection tracking memory utilization for 8M connections

Connection offered	nf conntrack buckets	HT load factor	Memory used (MB)
8 Millions	8388608	1	2560
8 Millions	4194304	2	2560
8 Millions	2097152	4	2560
8 Millions	1048576	8	2560
8 Millions	524288	16	2560
8 Millions	262144	32	2560
8 Millions	131072	64	2560
8 Millions	65536	128	2560
8 Millions	32768	256	2560

Finally, when the number of connections offered to connection tracking is larger than the nf_conntrack_max value, then the kernel would start to drop packet since there is no space left on the conntrack table to add the connection entry. Therefore, the threshold value of nf_conntrack_max should be a significantly larger value. Also, the physical memory of the system plays an essential part in storing the maximum number of connections. Therefore, the physical memory has to be sufficient enough to tolerate the larger amount of IP traffic.

4.1.3 Hash table load factor

The objective is to test the performance impact of the HT load factor for the nf_conntrack_max and nf_conntrack_buckets.

During the test, five different load factors (1, 2, 4, 8, and 16) were selected for analyzing the performance of the hash table. Also, load factor zero (i.e., without enabling connection tracking) has been tested for each millions connection, by enabling the NOTRACK in a prerouting RAW table.

The number of buckets (nf_conntrack_buckets) has been changed according to the nf_conntrack_max value during the test. For example, to create 8 million connections in the connection tracking table, the nf_conntrack_max was set to 8388608, and the nf_conntrack_buckets was 524288, which is load factor of 16.

Firstly, the connection states have been created by sending a particular number of IP packets, and the corresponding throughput is measured in packets per second (PPS). In addition to creating the connection, the created connection has been reused by sending the same IP traffic, and the corresponding throughput has been recorded as well. Each test has been run for 10 times in order to have more accuracy on the throughput value, and finally, the median value is drawn from these tests.

Figure 25 shows the PPS to create connection states in the conntrack table along with the PPS to reuse the connection state on different HT load factors.

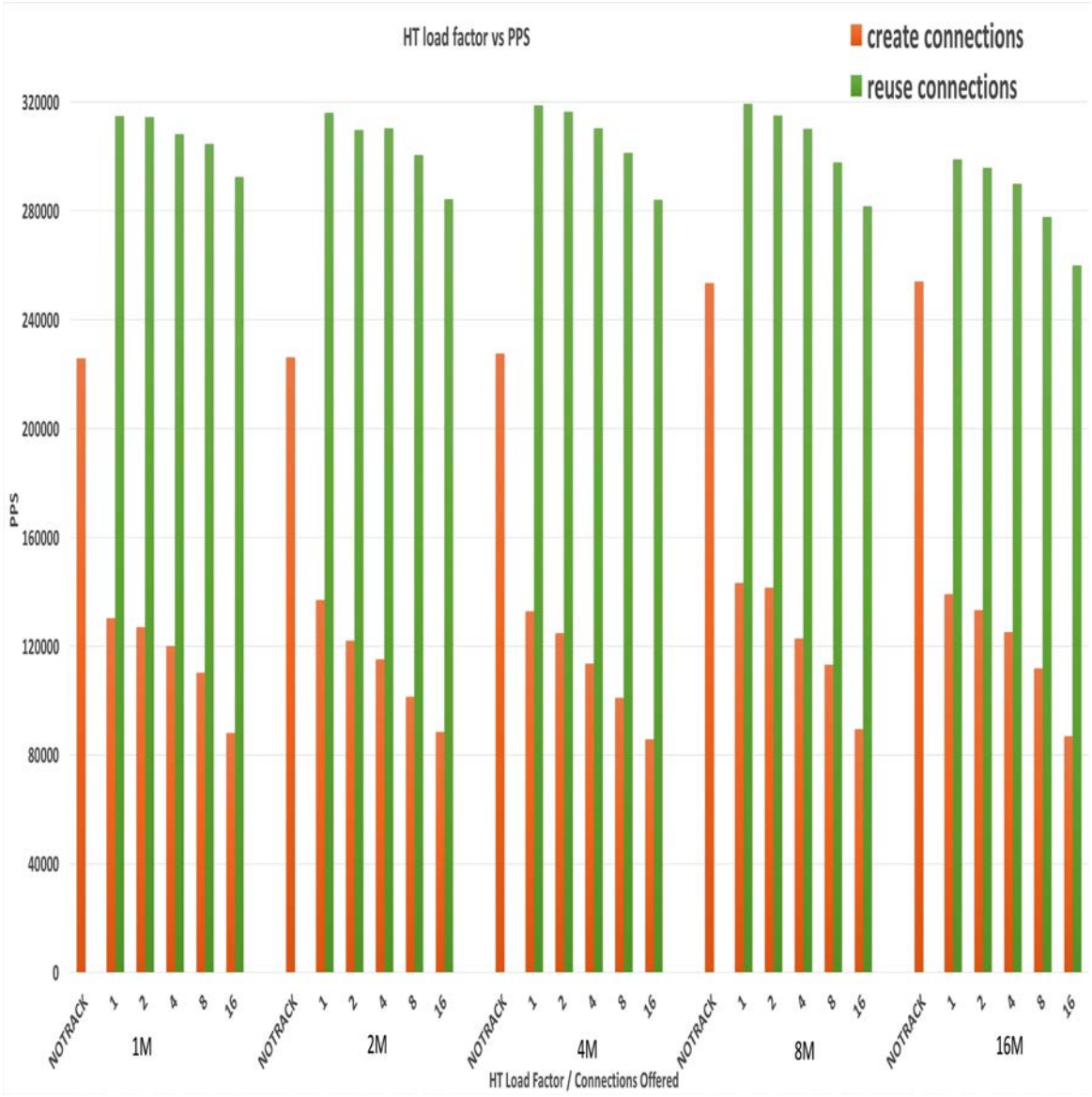


Figure 25: Millions of connections

Table 16 illustrates the throughput values for 8M connections, and it can be seen that the performance is better with the smaller HT load factor. Because of the fact that, with the smaller load factor, the length of the linked list in a bucket is also small. And iteration cost to add or find the position of the connection state entry in the bucket list depends on the length of the list. Therefore, the performance with smaller load factor is better.

Furthermore, without enabling connection tracking (NOTRACK) the performance is high for all number of IP packets as illustrated in Table 15. Because the connection states were not created and reused during the traffic flow.

Table 15: Connection tracking NOTRACK performance

Load factor(0)	Connections offered	Throughput(PPS)
NOTRACK	1M	225734
NOTRACK	2M	226244
NOTRACK	4M	227532
NOTRACK	8M	253452
NOTRACK	16M	254011

Table 16: Hash table load factor

Load Factor	nf conntrack max	nf conntrack buckets	Create throughput (PPS)	Reuse throughput (PPS)
1	8388608	8388608	143127	319298
2	8388608	4194304	141408	315009
4	8388608	2097152	122786	310210
8	8388608	1048576	113265	297796
16	8388608	524288	89405	281774

In addition, as HT load factor starts to increase, performance starts to drop. This is because the connection entry is stored in only one direction in a linked list (bucket), and hash value had to be calculated based on the packet information and indexed in a hash table. Further, iteration is done over the linked list to find an appropriate entry. Therefore, as the load factor increase, the length of the linked list in the hash table also increases, so the iteration time to find the entry also increases, thus decreasing the performance for creating and reusing the connections.

Finally, in conclusion, it is recommended to use the long hash table with a smaller linked list since the cost of hash calculation remains constant.

4.1.4 Thread scheduling on CPU

The primary objective is to observe the CPU performance on the different scenarios for thread scheduling in the CPU.

For analyzing the CPU performance, the HT default load factor, i.e., 8 and the load factor 16 were used, while the connection offered was 1 million connections. Also, the nf_conntrack_max has been configured to 4 million connections, in order to, use roughly 25% of the hash table. This configuration gives a consistent result as more elements in the hash table lead to the hash collisions.

To begin with, three test cases were identified for CPU performance test. Firstly, the normal OS handling for thread scheduling on the CPU, i.e., OS will decide which processes are executed on which core as well as managing the interrupts and the system calls.

The following command has been used during the normal CPU performance test. Secondly, the command taskset has been used for pinning the thread to a particular core of the CPU and the command used during the test is shown below. Finally, a specific core of the CPU has been reserved for the performance test.

```
tcpreplay –topspeed –loop = 1 –enable-file-cache –intf1 = source0s
1M_capture.pcap

taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cache –intf1 = source0s
1M_capture.pcap
```

During the test, IP traffic has been offered to the connection tracking, and the corresponding throughput has been documented. Next, for each test case; the connection states were created, and they were reused. These tests have been carried out 10 times to estimate the performance approximately. And finally, the median value was drawn from these 10 tests, which is illustrated in Table 17.

Table 17: Performance table for CPU scheduling

Load factor 16				
Test Case	**nf conntrack max**	**nf conntrack buckets**	**create(pps)**	**reuse(pps)**
CPU normal	4194304	262144	120265	162899
CPU pinning	4194304	262144	122850	199604
CPU reserve	4194304	262144	121581	197262
Load factor 8				
CPU normal	4194304	524288	126104	169166
CPU pinning	4194304	524288	135106	213130
CPU reserve	4194304	524288	129534	205508

It is observed that the CPU pinning and CPU reservation are giving more performance than the CPU normal because of the reason that the CPU was explicitly instructed to run a specified process on the particular core of CPU for both CPU pinning and CPU reservation. While, in the CPU normal, the OS has automatically handled the interrupt and system calls. However, the performance deviation was similar for all the three scenarios.

In addition, the CPU performance is identical for both the HT load factors 16 and 8, with insignificant variation. Figure 26 depicts the CPU performance for all test cases, and it is quite difficult to conclude which test case is performing better.

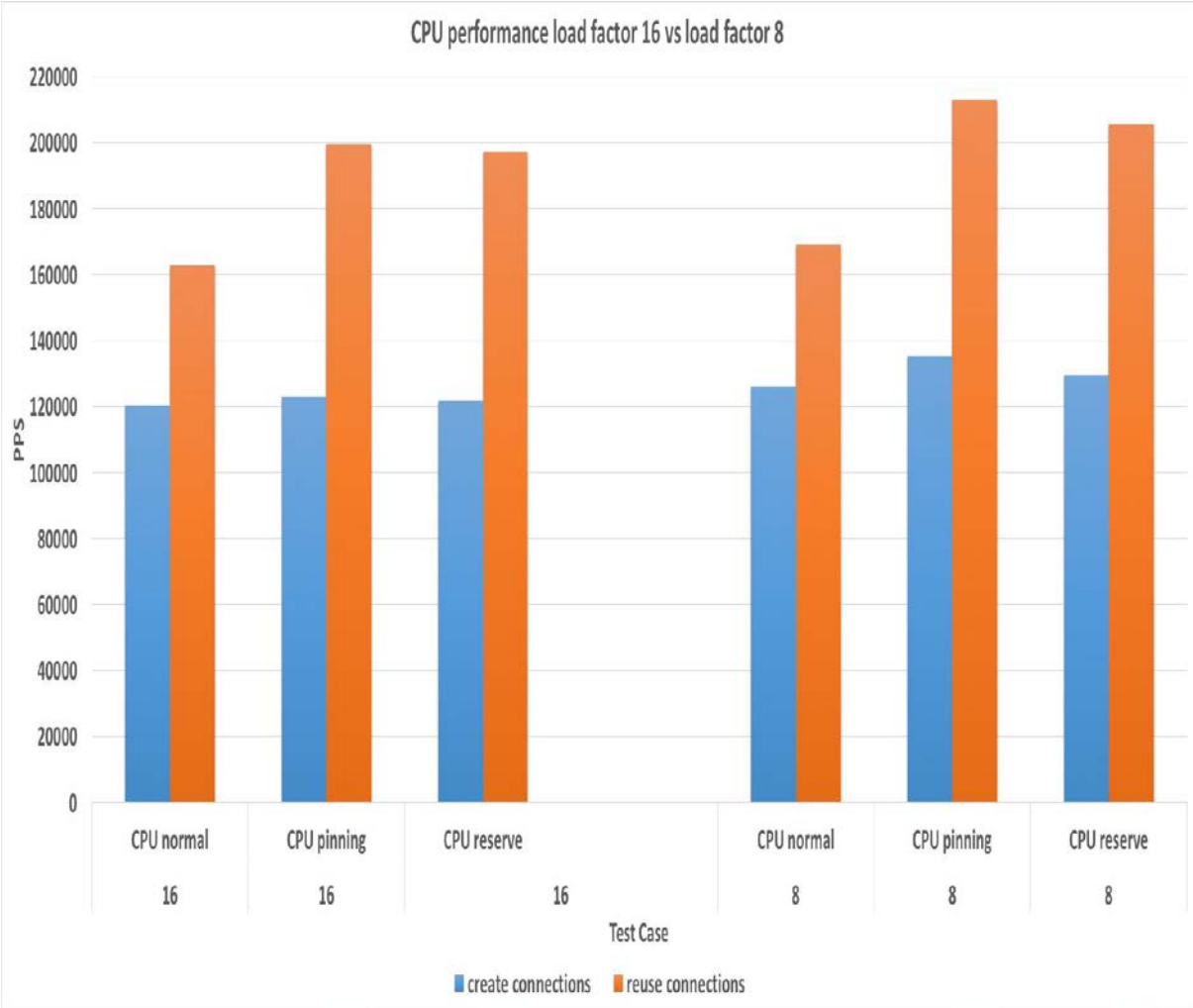


Figure 26: CPU performance

4.2 IPtables

In the Netfilter, shortly after the states have been created in connection tracking for IP packets, the IP packets proceed their journey within various tables in PRE-ROUTING chain and as well as in POSTROUTING chain. In different chains, the IP packets pass through the different tables, where different rules for IP packets are applied. Hence the operating system acts as a stateful firewall. In this section, different rules will be implemented on different tables in both PREROUTING and POSTROUTING chain to examine the performance of stateful firewall mechanism on Linux machine.

4.2.1 Iptables targets

The objective is to compare the performance of different IPTABLES targets.

Firstly, the nf_conntrack_max was set to 4 million (4194304), while the nf_conntrack_buckets was configured as 1048576, in order to create 1M connections. Therefore, only 25% of the hash table is used, to avoid any hash collisions. The following commands have been used.

```
sysctl -w net.netfilter.nf_conntrack_max = 4194304

sysctl -w net.netfilter.nf_conntrack_buckets = 1048576
```

Further, the performance test has been carried out by using five iptables targets, NOTRACK, FORWARD, DNAT, SNAT, and finally, SNAT/DNAT were carried out together. These targets were implemented mainly in the PREROUTING and the POSTROUTING chain. The NOTRACK, FORWARD, and DNAT were per-formed in PREROUTING chain, while SNAT in POSTROUTING chain and finally SNAT/DNAT in both PREROUTING and POSTROUTING chain. The command used to configure the different performance test scenarios are shown below.

```
iptables -t raw -I PREROUTING -i source0 -j NOTRACK

iptables -A FORWARD -i source0 -j ACCEPT

iptables -t nat -A PREROUTING -i source0 -j DNAT --to 100.255.255.254

iptables -t nat -A POSTROUTING -o sink0 -j SNAT --to 100.255.255.253
```

Secondly, after configuring the appropriate test scenarios, 1M connection states were created. After connection state has been established, IP packets are processed through the appropriate table(s) either in PREROUTING or POSTROUTING chain. Likewise, another 10 million packets (the original 1M packets merged 10 times) were offered for reusing established connection states. More-over, the test has been carried out 10 times to get more accuracy on the throughput data, and the median value is calculated for easier visualization.

In Figure 27, the performance of reusing the connection states is better than creating connection states. This is due to the fact that creating a new connection consume more time then reusing the same connection states. Further, NOTRACK is performing better than other iptables targets in terms of creating connections. Since the NOTRACK target will not add the state entry to the connection tracking. Therefore, the performance is much better than the other targets. While for the rest of iptables targets the performance remains overall similar. Although, SNAT performance is somewhat less as compared to other targets because SNAT is per-formed in the POSTROUTING chain. And the packets will be processed through each chain and tables within the chains before SNAT is carried out. Therefore, the performance is less than others targets.

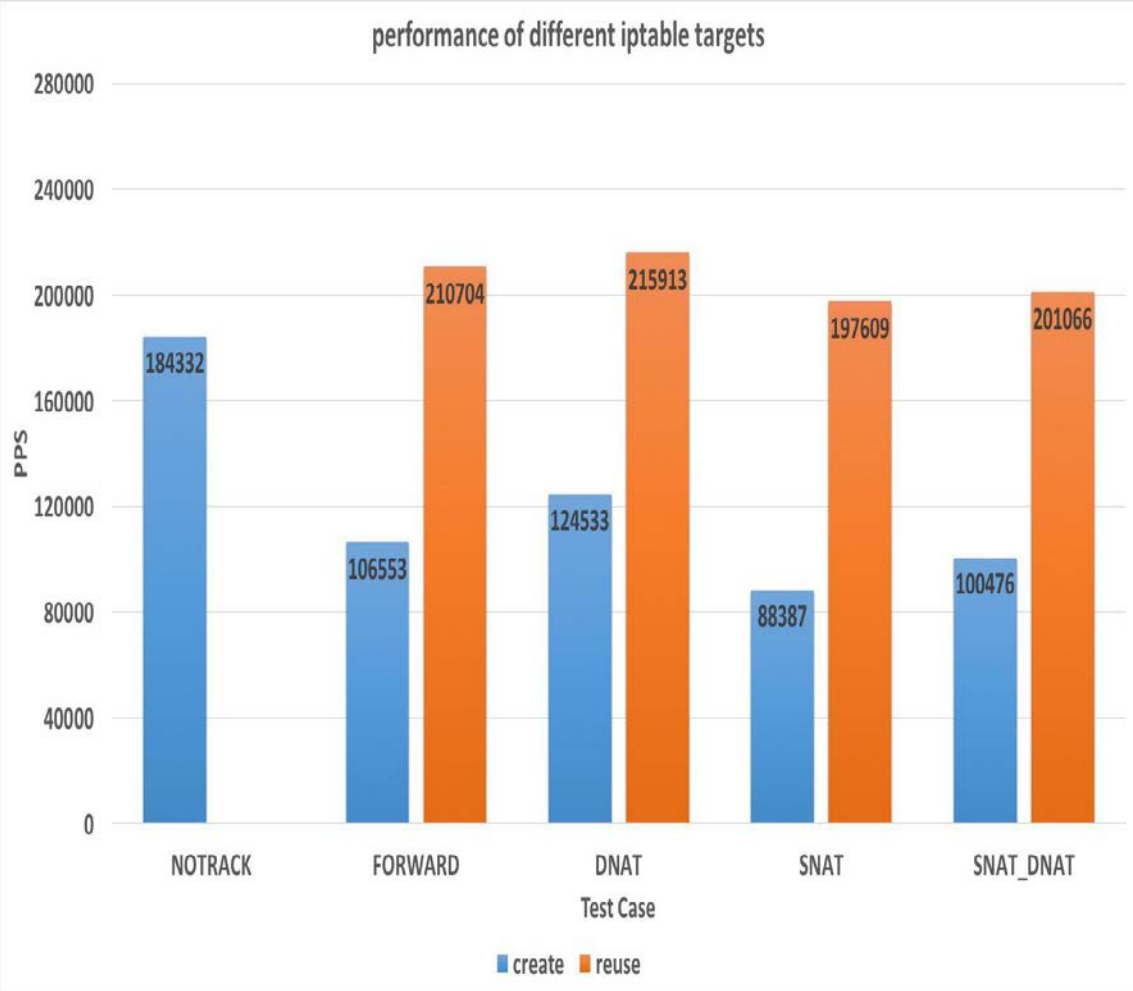


Figure 27: IPTABLES targets performance

4.2.2 Marking packets

The objective is to compare the performance of the iptables rules placed at the different positions in the NAT table, and matching the IP rules with packets that are marked in the MANGLE table.

Firstly, a separate chain named CIRCULAR_POOL has been added to the NAT table in the PREROUTING chain. Further, the 64K number of the IP rules have been attached to the CIRCULAR_POOL. And finally, the IP packets have been marked in the MANGLE table. And consequently, the IP packets traverse through the NAT table, which is performing DNAT to the marked IP packets.

The performance is measured by marking the IP packets to match the first, the middle and the last rule (i.e., rule 1, rule 32768 and rule 65536) in the NAT table. Then, the 64K IP packets were sent through the network to test the performance of the IP packets traversing through a large number of rules in the NAT table.

```
iptables -t nat -N CIRCULAR_POOL

iptables -t nat -I PREROUTING -m mark ! --mark 0 -j CIRCULAR_POOL

iptables -t nat -I CIRCULAR_POOL -m mark --mark 1 -j DNAT --to-destination 100.66.0.1

iptables -t mangle -A PREROUTING -p udp -j MARK --set-mark 1
```

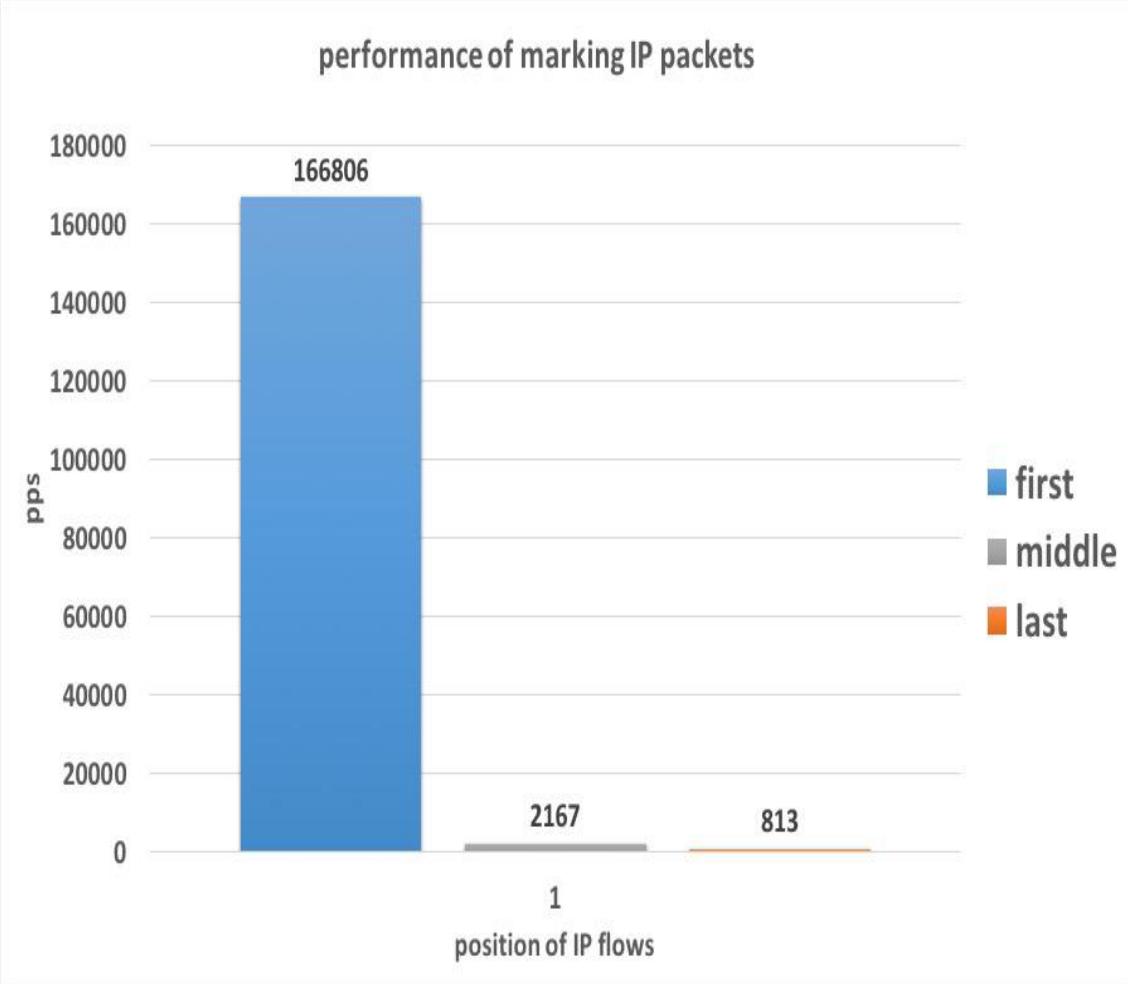


Figure 28: Performance for first, middle and last positions of IP rules

Figure 28, illustrates the vast performance difference between the different positions of the IP rules. The middle and the last position of the IP rule in the NAT table is giving the worst performance. Because each packet will traverse through all the rules in CIRCULAR_POOL until the match is found for the IP packet. Hence, the performance is very low for the middle and the last IP rule in the NAT table. Therefore, it is not appropriate to have a large number of rules in the NAT table.

Furthermore, a second test has been carried out to measure the performance of the 1st, 10th, 100th, 1000th and 10000th position of IP rule in NAT table, which can be seen in Figure 29. From the figure, it is understood that as the number of rules in the NAT table increases, the performance starts to degrade. There is significant performance degradation from the 100th position to the1000th position. Therefore, still having 1000 IP rules in the NAT table gives the worst performance.

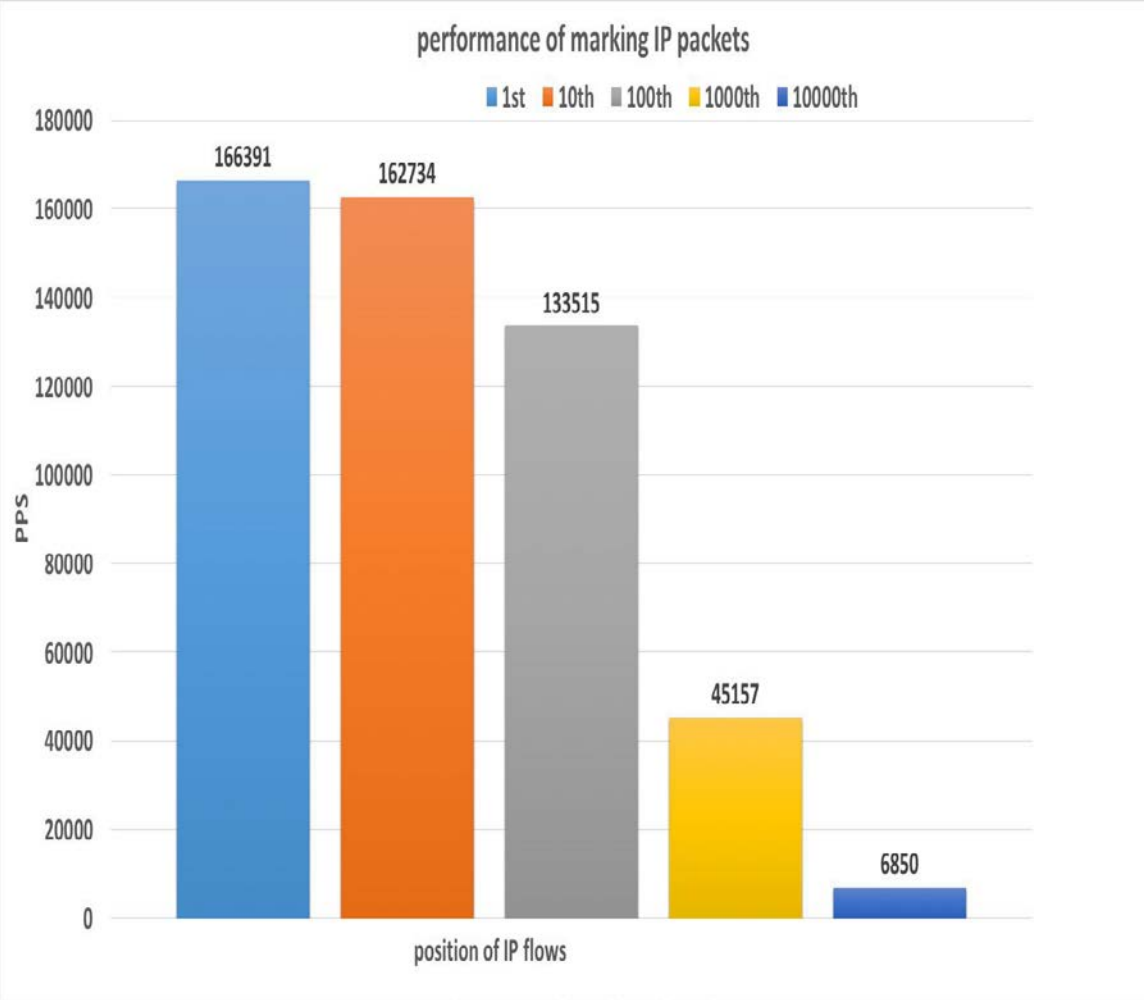


Figure 29: Performance for different positions of IP rules

Finally, to get an optimal number of IP rules in the NAT table, a third test has been carried out by testing the 1st, 10th, 100th 200th up to the 500th position in the NAT table and the result is shown in Figure 30.

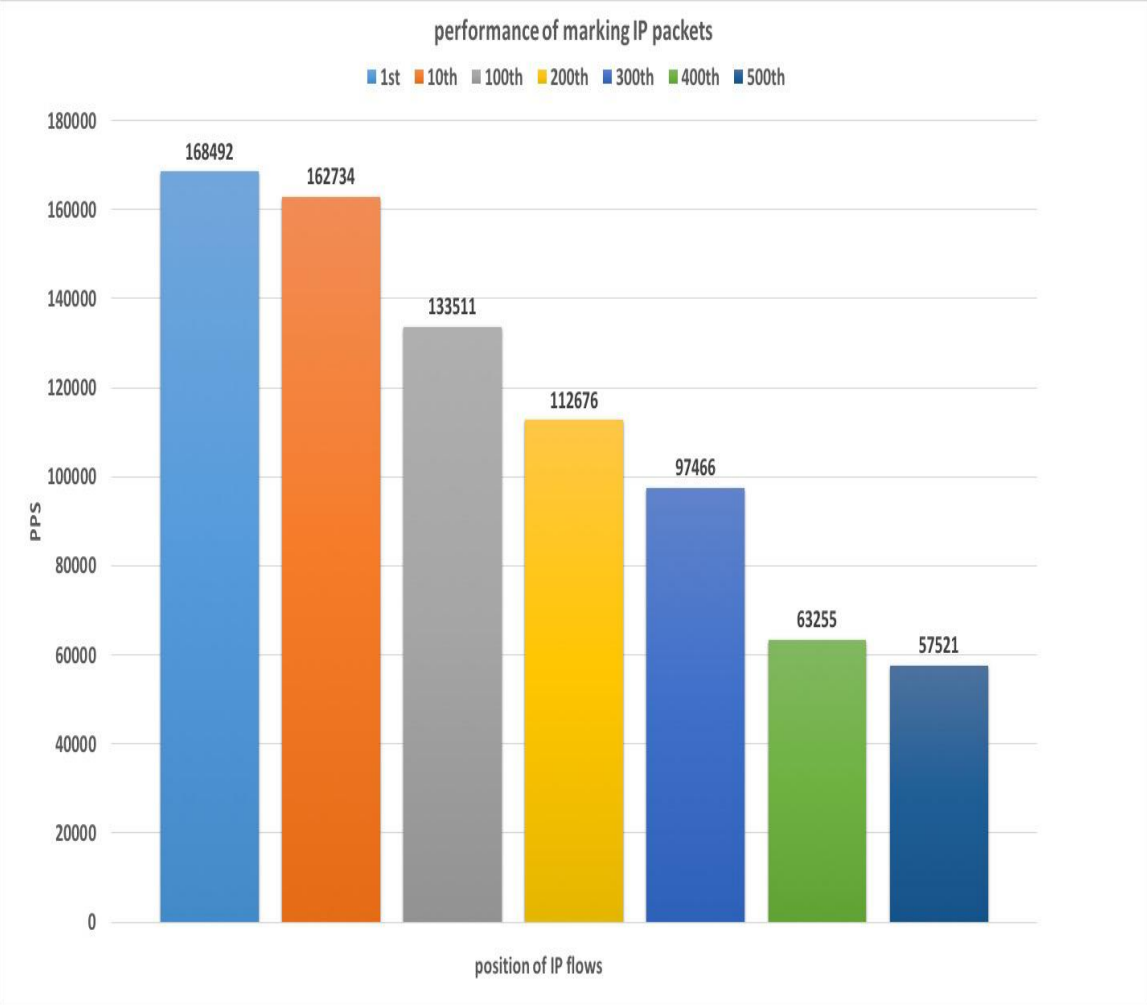


Figure 30: Performance for positions of IP rules

From Figure 30 it can be observed that in every 100 IP rule that is added; the performance is reduced by roughly 15%. Furthermore, the performance is quite acceptable until the 200th position of IP rule in the NAT table. Therefore, it will be best to configure iptables to have less than 200 rules in one table for getting the better performance.

4.2.3 Custom built iptables module

The primary objective of this test is to measure the performance of custom build module in iptables against the default behavior of the iptables.

During the test, a custom build iptables module named MARKDNAT has been installed. The custom module is the iptables extensions which requires the use of both a user and a kernel space module. Furthermore, the module can only be used in NAT table in the PREROUTING chain, which implements the standard action of the MARK and the DNAT target. Hence this module allows performing DNAT operation based on the packets mark. [5]

First and foremost, the test was carried out by designing 2048 IP rules in the NAT table, for translating the destination IP address of the packet. Next, the IP packets are marked in the MANGLE table of the PREROUTING chain. Then, the 1M packets were sent through the network to test the impact of the marking of the IP packets and its implications of traversing the IP packets through a large number of rules in the NAT table. Furthermore, another test was carried out, where only one rule known as the MARKDNAT was implemented in the NAT table. And finally, the same number of IP packets were transferred through the network to measure the performance.

```
iptables -t nat -N CIRCULAR_POOL
iptables -t nat -I PREROUTING -m mark ! --mark 0 -j CIRCULAR_POOL
iptables -t nat -I CIRCULAR_POOL -m mark --mark 1 -j DNAT --to-destination
100.64.0.255
iptables -t nat -I CIRCULAR_POOL -m mark --mark 2 -j DNAT --to-destination
100.64.0.255
```

Commands for custom build module of iptables:

```
iptables -t     nat -N   CIRCULAR_POOL
iptables -t     nat  -I  PREROUTING -m mark ! --mark 0 -j CIRCULAR_POOL
iptables -t     nat  -I  CIRCULAR_POOL -j MARKDNAT --or-mark 1  -m comment
--comment    'DNAT to packet mark'
```

Figure 31 illustrates the performance difference for the default behavior of iptables against the custom built module in iptables. From Figure 31, it is seen that as the position of rule increases, the performance starts to decline for the default behavior of the iptables. Further, the performance degradation is due to the time consumed by the iptables for finding the appropriate match on the IP rules in the NAT table. Therefore, performance is directly proportional to the number of IP rules in the NAT table. At the same time, the performance of the custom build module of the iptables remains constant across any number of the IP rules in the NAT table. Hence, implementing custom built iptables module eliminates the bottleneck and increases performance. And finally, simplify the implementation of IP rules in the NAT table.

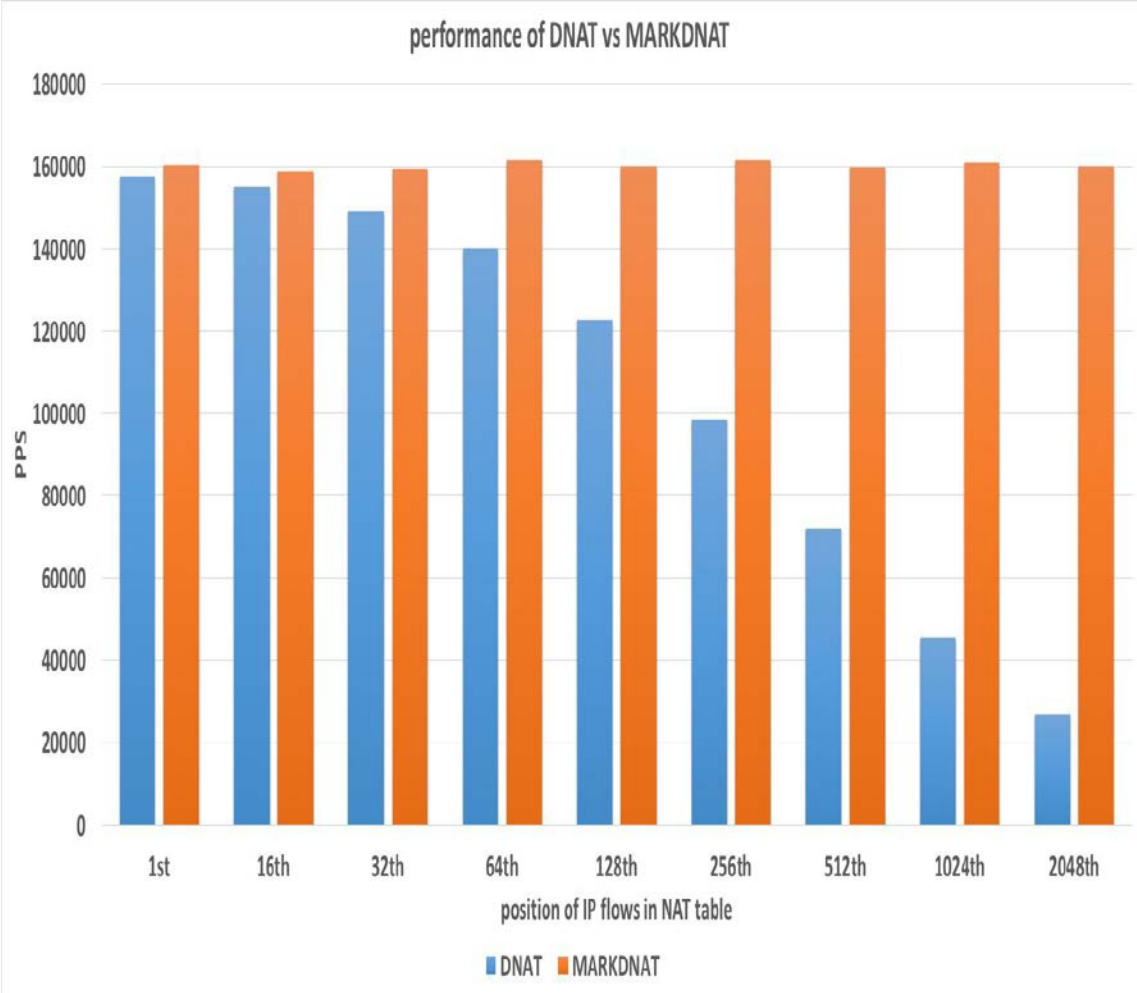


Figure 31: performance of custom MARKDNAT

In conclusion, it is not recommended to have a large number of IP rules in the iptables. Since a large number of rules significantly slows down the performance. And, in order to get the optimal performance, MARKDNAT module should be implemented.

4.2.4 NFQUEUE target

The objective is to test the performance of Netfilter kernel module to create multiple queues and finally, process the queued IP packets at user space.

The NFQUEUE target is mainly used for queuing the IP packets in the kernel space to be processed by user-land applications before sending the IP packets to the final destination. NFQUEUE provides a mechanism for passing the IP packets out of the stack for queuing to user space. And, then receiving the IP packets back into the kernel with a verdict specifying what to do with the IP packets (such as ACCEPT or DROP). Further, the IP packets may also be modified in user space before reinjection back to the kernel.

In order to carry out the test, a custom builds Netfilter-nfqueue module has been compiled with source code from the git hub. [24] Next, using the source code, four and eight queues were created to test the performance for 1M packets. In addition, different test cases were proposed to test the performance of the NFQUEUE, for instances queue-CPU-fanout, without using CPU-fanout, in the simple forwarding state and iptables performing DNAT. Following commands have been executed to create four and eight queues.

```
iptables -t mangle -A PREROUTING -i source0 --src 10.3.0.1/18 -j NFQUEUE --queue-num 0

iptables -t mangle -A PREROUTING -i source0 --src 10.3.64.1/18 -j NFQUEUE --queue-num 1

iptables -t mangle -A PREROUTING -i source0 --src 10.3.128.1/18 -j NFQUEUE --queue-num 2

iptables -t mangle -A PREROUTING -i source0 -j NFQUEUE --queue-balance 0:3 --queue-cpu-fanout
```

Figure 32 illustrates the performance of NFQUEUE for the different scenarios, for instance, simple forwarding, while using CPU fanout and no-CPU fanout options, and finally test with DNAT. It is seen that simple forwarding performance is better than DNAT because more processing is needed at prerouting chain after receiving packets from user space (i.e., modify destination address). Also, while using queue-CPU-fanout options, the load is spread across different queues. However, performance for using queue-CPU-fanout option and without using fanout options remains similar for all test cases.

Moreover, the nfqueue source code is compiled to test for eight queues. As compared to the queues four test, the performance is somewhat lower because the new test case has a large number of queues and it takes time for processing all queues. Therefore, it could be concluded that lesser number of queues should be implemented for achieving higher performance.

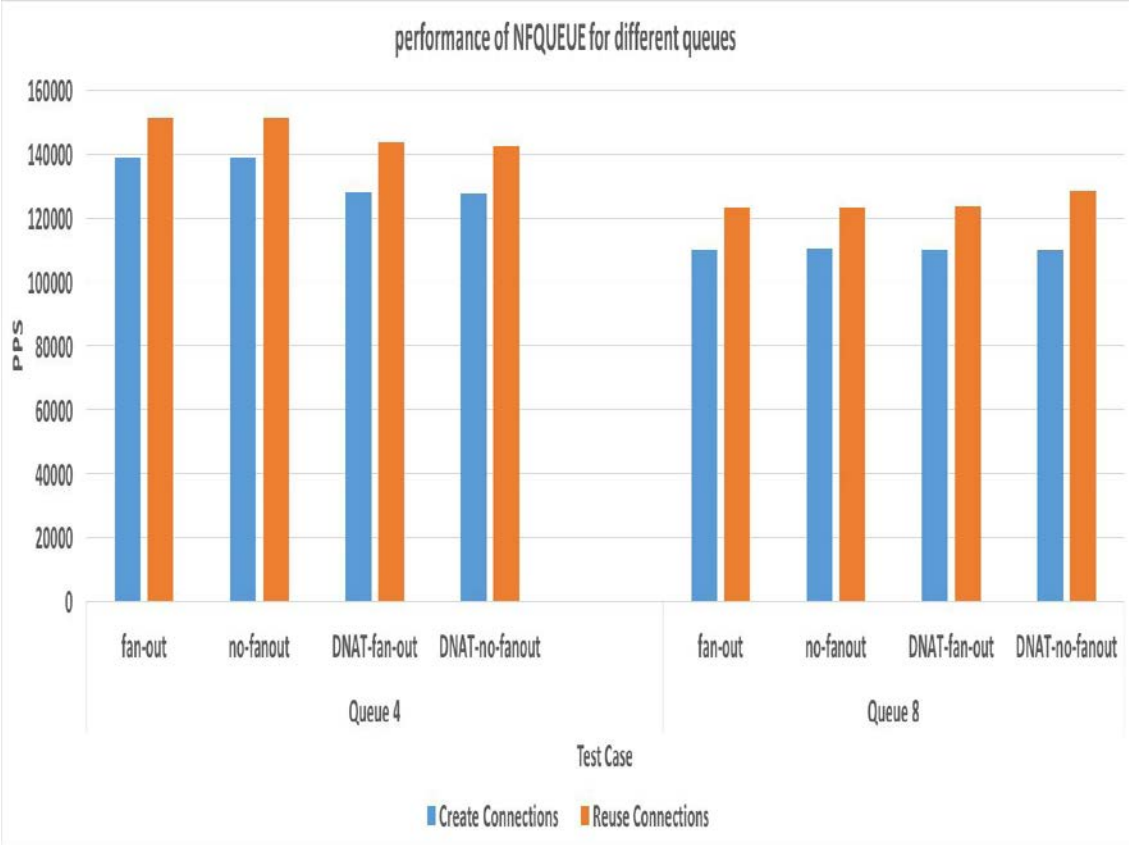


Figure 32: Tcpreplay performance for nfqueue

4.3 Designing optimal architecture for IP flows/rules

The primary objective in this section of testing is to determine a model/design for arranging a large number of IP rules/flows in the different chains. The design will be identified as the best-performing model based on the hit-counts; lower the hit-counts, the better the design for arranging the IP rules/rules. Further, the development of a model includes developing a mathematical formula for determining the total hits-counts for all number of IP rules/flows. After developing a mathematical formula, a specific number of rules will be arranged in the different designs. During the testing phase, both the linear and the nested model will be tested mathematically and practically.

4.3.1 Linear arrangement of flows/rules

The objective of the test is to develop a mathematical formula for arranging the different number of IP rules/flows in the linear architecture and based on the formula, testing the performance of different linear architectures.

In the linear architecture, chains will be arranged linearly and IP rules/flows are added to corresponding chains, so that the matching of IP rules will be processed from top to down until a match is found. Therefore, for the 64000 number of IP rules, there could be a different number of linear architectures for arranging the rules. The mathematical model, equation 1, try to design different architectures based on the total hit counts for a particular design. The formula for determining a total hits as per the number of IP rules in a specific chain(s) on the iptables.

$$Hitcount\ (C) = \sum_{i=1}^{S} (R_i * i + \sigma_i) \tag{1}$$

Where, **S** - is the number of selectors or chains.

$\mathbf{R_i}$ - is numbers of IP rules/flows in $\mathbf{i^{th}}$ selectors

$\boldsymbol{\sigma_i}$ - is sum of numbers of IP rules in $\mathbf{i^{th}}$ selectors,

$$\sigma_i = \sum_{j=1}^{r} (j) \tag{2}$$

Equation 1 represents a formula for the linear arrangement of IP rules/flows in different chains. And, the total hits-count can be determined for all the number of rules/flows in a linear design. Based on equation 1, Table 18 illustrates different numbers of the linear model and the corresponding number of total hits.

Table 18: Linear design

Number of chain(s)	Number of rules per chain(s)	Total hit counts
1	65536	2147581952
2	32768	1073872896
4	16384	537067520
8	8192	268763136
16	4096	134807552
32	2048	68222976
64	1024	35717120
128	512	21037056
256	256	16842752

As seen from Table 18, there will be less hits, when both the number of IP rules and the number of chains are equal. The number of IP rules in chains determines the number of hits and larger the number of IP rules yields the larger numbers of hits so that performance will be lower and vice-versa. Therefore, for optimal performance, it is recommended to have both the number of chains and IP rules in a chain to be the same or nearly equal, as concluded from the mathematical model.

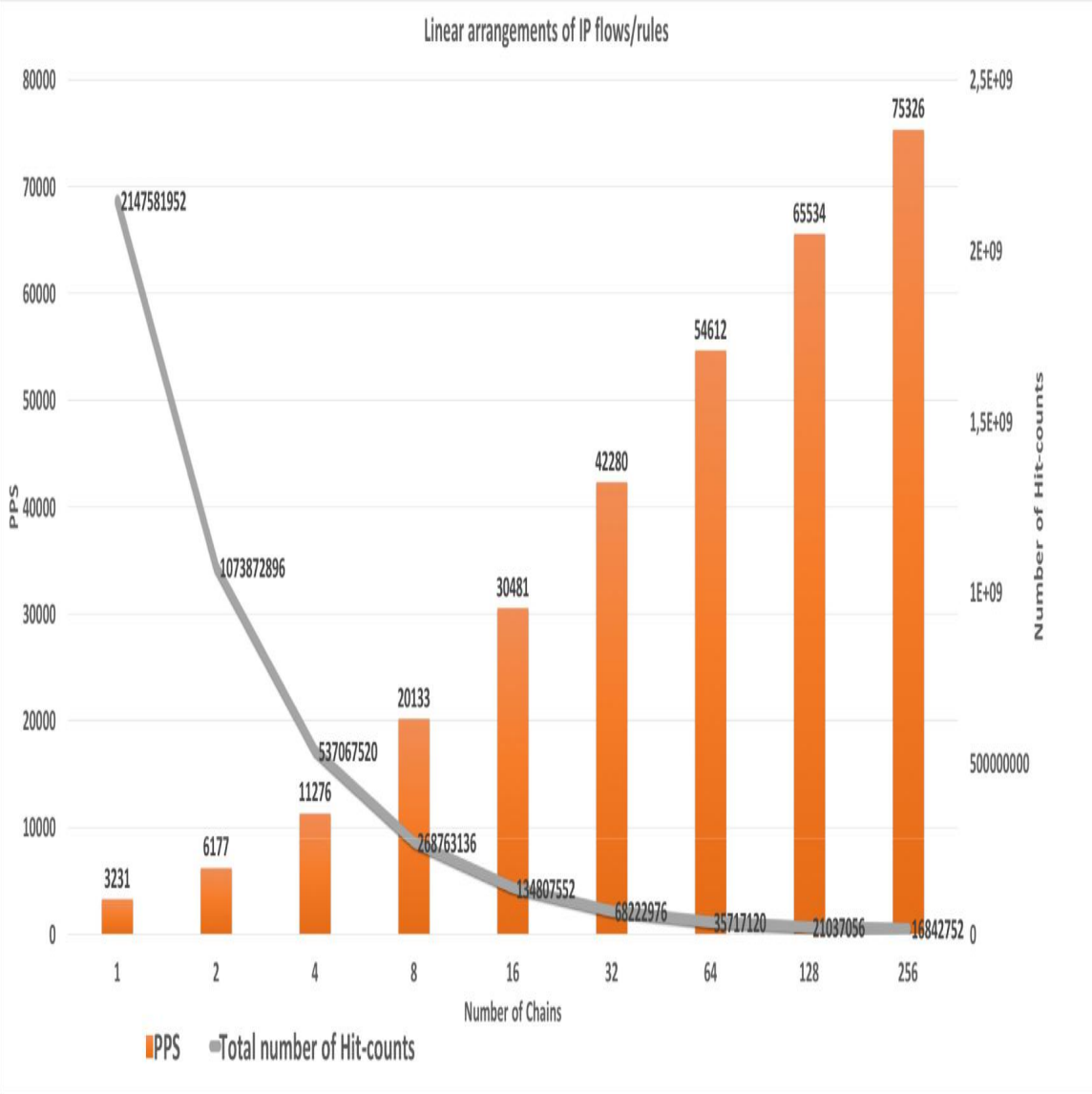


Figure 33: Tcpreplay performance or linear-design of IP flows/rules

Figure 33 illustrates the performance of different linear design of IP rules/flows as per the mathematical model. It is seen that as the number of chains increases and the number of IP rules decreases, performance starts to increase, whereas the number of hit starts to drop. The best performance is yielded, when the number of chains and the number IP rules in a chain are equal, which has been confirmed by the mathematical model. Therefore, in conclusion, while implementing the IP rules in iptables it should be noted that the number of chains and the number of IP rules in a chain should be the same or nearly equal for better performance.

4.3.2 Multi-step selectors for arranging flows/rules

The objective is to design a mathematical formula to minimize the hit-counts across multiple chains and finally, test the performance by arranging the IP flows in a nested multi-step chains architecture.

In the nested design, the chains will be arranged in a tree-like nested arrangement as shown in Figure 34. Figure 34 illustrates the basic idea of how nested chains are designed and how the IP packets jump through different branches of the chain. Each chain will have a certain number of generic IP flows/rules which will be further divided into sub-chains with more specific IP flows/rules. For example, in Figure 34, the first chain/selector will have IP flow that will have a match for all the IP packets. Next, the second level selector will have IP flows that have a match for all the IP packets for that specific branch only. And finally, the third level will have only one IP flow that matches for the single IP packet flow only.

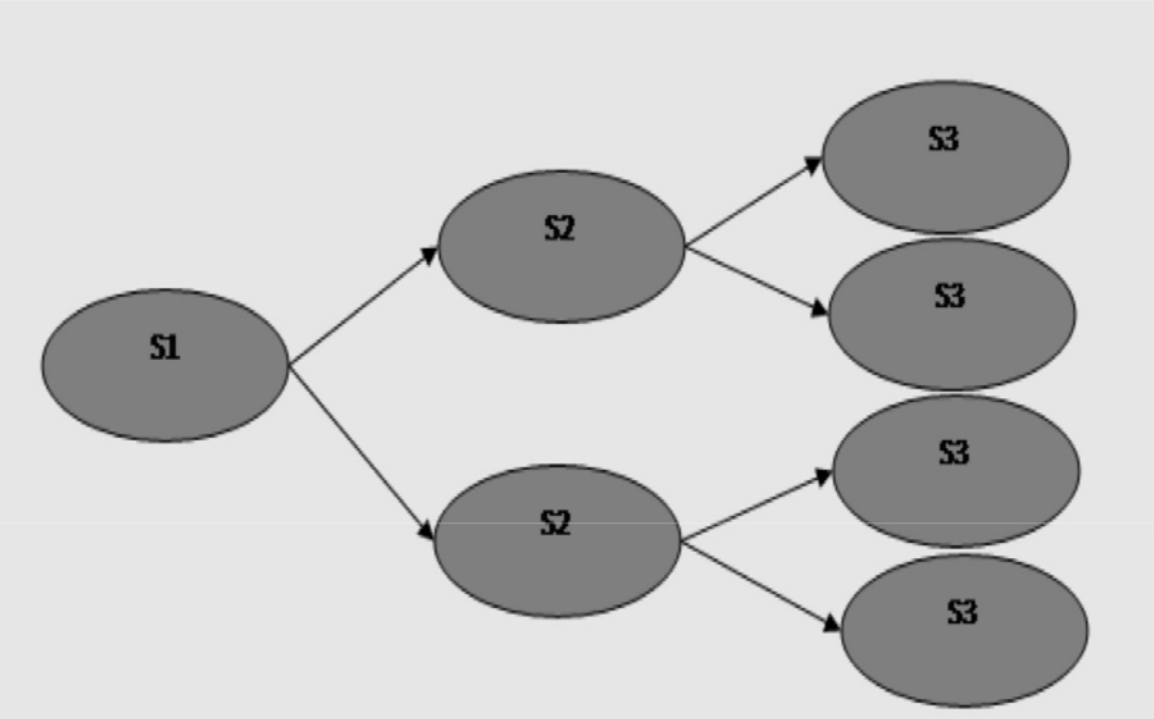


Figure 34: example of nested design of flows

Therefore, the IP packets will hit the first selector and based on the IP flows in the first selector. The IP packets jump to the next level selector within the same branch until a match is found.

Hence, for arranging the different numbers of rules, there could be many different arrangements. Equation 3 try to design the different mathematical model based on total hit-counts. The optimal design will be one with least hit-count.

The formula for determining the total hit-count per the number of IP rules in the chain(s).

$$Hitcounts\ (C) = \beta * R + Ft * \sum (R) \tag{3}$$

Where, **Ft** – is number of final selectors/chains.

R – is the number of IP flows in that selector(s)/chain(s).

β – is the sum of all possible combinations of selectors in different level.

For example, in three level of selectors β would be the following:

$$\beta = \sum_{i=1}^{S1} + \left\{ \sum_{j=1}^{S2} + \left\{ \sum_{k=1}^{S3} (i + j + k) \right\} \right\} \tag{4}$$

Where,

S1 - is first level of selectors,

S2 - is second level of selectors, and

S3 - is third level of selectors.

From the equation 3, nested arrangement of the IP flows can be determined along with the total hit-counts for all the IP flows in the particular design. Furthermore, based on the equations 1 and 3 the following four different architectures have been designed to test the performance of the 64000 number of IP flows.

– Level_1_selectors: It has total 256 number of chains, and each chain has 256 rules. This architecture is designed based on equation 1, while rest of the other architectures are based on equation 3.

– Level_2_selectors: This is arranged in two level of selectors; the first level has 16 chains, and each chain is further divided into 16 sub-chains in the second level. Therefore, the final number of chains is 256, and each chain has 256 rules.

– Level_3_selectors: It consists of three level of selectors. The first level has four chains, and each chain has further eight sub-chains in the second level. And finally, each chain in the second lever are further divided into eight chains at the level three. Therefore, the final number of chains is 256, and each has 256 rules.

– Level_4_selectors: Four level of selectors form this architecture. The first level has four chains, and subsequently, each level of selectors also has 4 sub-chains and so on until the level of four. Therefore, the final number of chains is 256, and each chain has 256 rules.

In addition, Table 19 illustrates the different numbers of level of selectors and the corresponding total hit-counts.

Table 19: Nested design of IP flows/rules

Level of selectors	Number of final chain and rules	Total hit counts
1 [Linear]	256, 256	16842752
2[s1=16, s2=16]	256, 256	9601024
3 [s1=4, s2=8, s3=8]	256, 256	9240576
4 [s1=4, s2=4, s3=4, s4=4]	256, 256	9142272

Furthermore, from Table 19 it could be visualized that as the nesting of chains increases, the total hit-count decreases. The linear design is the worst architecture in terms of hit-counts since it has the highest of hit-counts (more than 80% of hit-counts) as compared to other designs. While other three models are quite similar to each other, only minor variation in terms of hit-counts. Among the four designs, the four-layered nested architecture has the least hit-count because when the number of nested steps/levels increases, the possibility of finding a match in the chains decreases, and the hit-counts is also reduced. Besides, the nesting of chains also provides more controlled management of IP flows in the chains. Since the IP flows with the more specific match could be arranged carefully, to have the least hit-counts.

Finally, the test setup has been designed as per the mathematical model, where chain(s) and IP flows/rules are arranged accordingly to Table 19. Finally, the test has been carried out to measure the performance of different designs. Finally, with the performance data and the hit-counts, a graph has been plotted as shown in Figure 35.

Figure 35 conforms to the mathematical model of nested design for arranging IP flows. It is seen that level-4 design is the best design as compared to others. However, the performance is not heavily different among nested designs, since the hit-counts are similar. Therefore, it can be concluded that the hit-counts can be reduced by arranging the IP flows in multi-step chains. As the number of steps/levels in design increase, so performance also increases. Consequently, nesting of selectors is more appropriate than a linear arrangement of IP rules/flows, since it gives more controlled management of rules and chains as well as better performance.

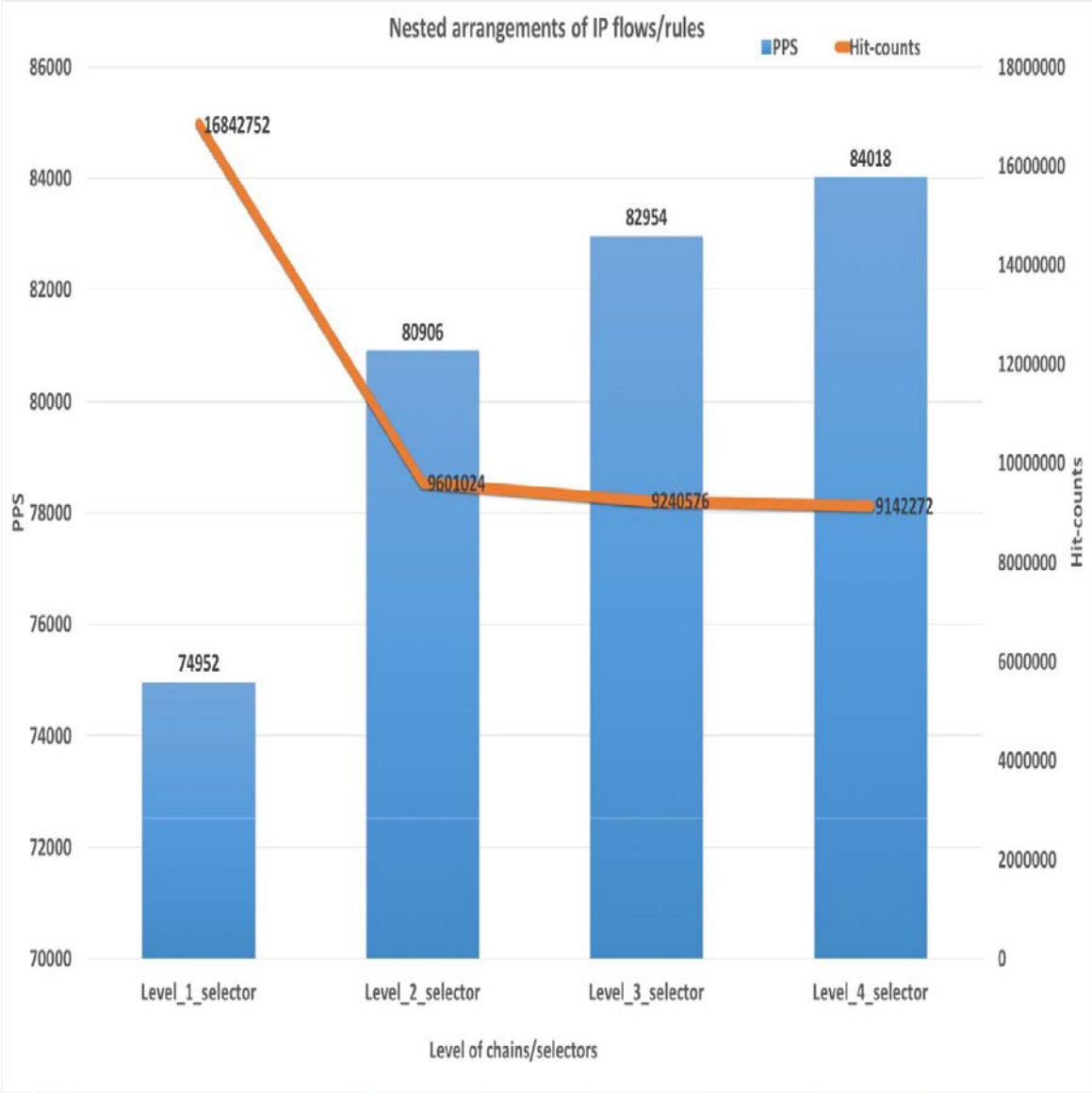


Figure 35: Tcpreplay performance for nested-design of IP flows

5 Conclusion and Discussion

The primary objective of the thesis was to evaluate the forwarding performance of Linux Netfilter subsystem in order to deploy the PRGW on top of Linux architecture to ensuring that all components of PRGW would operate as designed. And finally, investigate the scalability issues of PWGW for provisioning a node for millions of subscribers. Furthermore, this thesis evaluates the performance of Netfilter by using different parameters for instance, how many simultaneous connections can be created, physical memory consumed by connections, overloading CPU process, IP packets filtering and packets mangling.

At the beginning of thesis work, I was completely unaware of the networking stack, how it operates and how to test the performance of Netfilter subsystem. However, with the significant amount of studying and assistance from my advisor, my skills on testing of Netfilter systems grew. This thesis has been a skills learning experience for me as I was able to learn many new technologies and their implementations.

During the performance testing, each test objectives and results were predicted based on the theoretical knowledge, which simplified the testing process. The results obtained were very close to the objectives that were set before starting the tests, which has been explained in the test and evaluation section. The outcome of this thesis has been accomplished as proposed in the objective section. In order to reach the goal, physical Linux server has been selected, for the accuracy of the testing and performance evaluations. The performance is evaluated based on the throughput, which is measured in packet per seconds (PPS).

Firstly, in the connection tracking, the primary objective was to scale as many numbers of simultaneous connections as possible. In order to observe the behavior of Linux machine that is acting as a router. The behavior of a device includes the physical memory usages, system limitations or bottlenecks.

In this thesis, we have observed that it is possible to scale millions of simultaneous connections in connection tracking. In fact, during the performance testing, 16M connection states were stored in connection tracking. Furthermore, we have concluded that it is possible to scale to as many simultaneous connections as the number supported by nf_conntrack_max and the nf_conntrack_buckets parameters in Linux kernel. Since there is a hard-coded limit on both the parameters and beyond these limits system would not accept any values.

Apart from the system limits, physical memory is another major issue as we have observed that the different number of connection states occupy a certain amount of physical memory. For example, in the performance test, we have seen that 16M connection states occupy 5210MB of physical memory. And if the machine did not have enough memory space to store these connection states, the system would crash because the system would run out of the physical memory.

Secondly, the performance of different iptables targets has been measured, to visualize the stateful packet filtering/mangling mechanism in Linux machine. During the IP packets filtering/mangling tests, we have measured the performance of default behavior of different iptables targets, and compare with the performance of custom built module for IP packets filtering/mangling. Furthermore, we have concluded that the custom module provided the higher performance than the default iptables targets for packets filtering/mangling. In addition, we have also visualized the performance of adding a large number of IP flows/rules to the iptables in different chains and tables. We conclude that a large number of IP flows/rules in a single table degrade the performance.

And finally, in this thesis, we have designed a mathematical formula as well as different system designs for arranging a large number of IP flows/rules for getting the higher performance. One of the system designs includes linearly arranging of IP flows in different iptables chains. While another design is to nest the chains in iptables and add the IP flows to these nested chains. Further, we have concluded that for getting the better performance we have to arrange IP flows/ rules in a nested chain design.

In conclusion, this thesis has evaluated the viability of building PWGW a high-performance IP router built on top of Linux operating system. The different experimental tests were carried out to measure different forwarding rates of Netfilter in different scenarios to conclude that PRGW can be deployed over the Linux architecture. Finally, this thesis has tried to give a contribution by reporting results obtained from the testing of Linux networking stack.

References

[1] J. L. Santos. Private realm gateway. Master's Thesis, Department of Communications and Networking, Aalto University,Espoo, 2012.

[2] R. Kantola, J. L. Santos, N. Beijar Policy-based communications for 5g mobile with customer edge switching. Security and Communication Networks, Depart-ment of Communications and Networking, Aalto University,Espoo, 2015.

[3] Routing Edge-to-Edge and Through Ethernets. http://www.re2ee.org/

[4] R. Kantola, J. L. Santos, N. Beijar and P. Leppäaho. Implementing NAT Traversal with Realm Gateway. IEEE Int'l Conference on Communications, 2013.

[5] J. L. Santos. customer_edge_switching_v2. https://gitlab.cloud.mobilesdn.org/CES/customer_edge_switching_v2

[6] S. Guha, K. Biswas, B. Ford, S. Sivakumar and P. Srisuresh. NAT Behavioral Requirements for TCP. https://tools.ietf.org/html/rfc5382

[7] Cisco Inc. IP addressing: NAT configuration Guide, IOS relese 15M&T. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_nat/ configuration/15-mt/nat-15-mt-book/iadnat-addr-consv.html

[8] K. Egevang and P. Francis, The IP Network Addresses Translator (NAT). https://www.ietf.org/rfc/rfc1631.txt

[9] P. Ellingwood, IP Network Address Translator (NAT) Terminology and Considerations. https://tools.ietf.org/html/rfc2663

[10] J. Srisuresh, A Deep dive into Iptables and Netfilter Architecture. https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture

[11] P. N. AYUSO, Netfilter's connection tracking system https://people.netfilter.org/pablo/docs/login.pdf

[12] Wikipedia, Flow of network packets through the Netfilter https://en.wikipedia.org/wiki/Netfilter#/media/File:Netfilter-packet-flow. svg

[13] R. Russel and H. Welte, Linux Netfilter hacking HOWTO. https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.html#toc3

[14] M. Bonola, L. Bracciale, Packet filtering with Linux, Netfilter and IP tables. http://www.grep.it/RMD/05-Netfilter.pdf

[15] M. Boye, Netfilter Connection Tracking and NAT Implementations. https: //wiki.aalto.fi/download/attachments/69901948/netfilter-paper.pdf

[16] Man page of IPTABLES Man page of IPTABLES. http://ipset.netfilter. org/iptables.man.html

[17] Iptables info. Iptables. http://www.iptables.info/en/iptables-contents. html

[18] VOIP magazine. Tuning the Linux connection tracking system. https://voipmagazine.wordpress.com/2015/02/27/tuning-the-linux-connection-tracking-system/

[19] Ostinato org. Ostinato User manual https://userguide.ostinato.org/

[20] TCPDUMP. tcpdump(8) - Linux man page https://linux.die.net/man/8/ tcpdump

[21] TCPREPLAY. tcpreplay http://tcpreplay.synfin.net/wiki/tcpreplay

[22] tcpreplay(1) - Linux man page. tcpreplay https://linux.die.net/man/1/ tcpreplay

[23] TCPREWRITE. tcprewrite(1) - Linux man page https://linux.die.net/ man/1/tcprewrite

[24] NFQUEUE Source Code. netfilter-nfqueue-samples https://github.com/ irontec/netfilter-nfqueue-samples

A Appendix

Script for analysis of Connection tracking and hash table #!/bin/bash

```
if [[ $UID != 0 ]]; then
        echo "Please run this script with sudo:"
        echo "sudo $0 $*"
exit 1
fi
#this script is for testing the Conenction tracking and Hashtable load factor.
#for finding optimal performance, only loadfactor from 1 through 64 has been selected if [ -z "$1" ];
then
        echo usage: $0 [nf_conntrack_max] [pcap-file to create connections] [pcap-file to reuse connections] echo

        echo e.g. $0 2097152 1M_capture.pcap 10M_merge_capture.pcap echo
exit
fi
MAX_CONN=$1
PCAP_CREATE=$2
PCAP_REUSE=$3
LOGFILE_CREATE="create.log"
LOGFILE_REUSE="reuse.log"
echo "Removing previous log files"
rm $LOGFILE_CREATE
rm $LOGFILE_REUSE
echo "changing the connection tracking maximum value to be $1"
echo "..."
sysctl -w net.netfilter.nf_conntrack_max=$1
sleep 2
#The test is run 10 times each HT loadfactor
for a in 1..10
do
echo "*** iteration #$a ***" » $LOGFILE_CREATE
echo "*** iteration #$a ***" » $LOGFILE_REUSE
echo "**********************"
for i in 1 2 4 8 16 32 64
do
        BUCKET=$(($MAX_CONN/$i))
        echo "*** Load factor #$i ***" » $LOGFILE_CREATE
        echo "1.Changing bucket size to $BUCKET" » $LOGFILE_CREATE
        echo "....................."
        sysctl -w net.netfilter.nf_conntrack_buckets=$BUCKET
        echo "2. Flushing conntrack"
        conntrack -F
        sleep 2
        echo "3. Creating new connections in conntrack"
        echo "....................."
        taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cache –intf1=source0s                $PCAP_CREATE » $LOG-
FILE_CREATE
        sleep 2
        echo "*** Load factor #$i ***" » $LOGFILE_REUSE
        echo "Bucket size is $BUCKET" » $LOGFILE_REUSE
        echo "4. Reusing connections in conntrack"
        taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cache –intf1=source0s $PCAP_REUSE » $LOGFILE_REUSE
        echo "**************************"
        done
echo "**************************"
done
```

Script for testing the CPU performance

```
#!/bin/bash
#   Add this at the beginning of the script to assure you run it with sudo if [[ $UID
!= 0 ]]; then
        echo "Please run this script with sudo:" echo
        "sudo $0 $*"
```

```
        exit 1
fi
if [ -z "$1" ]; then
        echo usage: $0[pcap-file to create connections] [pcap-file to reuse connections] echo
        echo e.g. $0 1M_capture.pcap 10M_merge_capture.pcap
        echo
                exit
fi

PCAP_CREATE=$1
PCAP_REUSE=$2

LOGFILE_CREATE="cpu_normal.create.log"
LOGFILE_REUSE="cpu_normal.reuse.log"
echo "Removing previous log files"
rm $LOGFILE_CREATE
rm $LOGFILE_REUSE

for i in {1..10}
do
        echo "*** Starting iteration #$i ***"
        echo "1. Flushing conntrack"
        conntrack -F
        echo "2. Creating new connections in conntrack"
        tcpreplay –topspeed –loop=1 –enable-file-cache –intf1=source0s $PCAP_CREATE »
        $LOGFILE_CREATE
        echo "3. Reusing connections in conntrack"
        echo "*** iteration #$i ***" » $LOGFILE_REUSE
        tcpreplay –topspeed –loop=1 –intf1=source0s $PCAP_REUSE » $LOGFILE_REUSE
done
```

Script for testing the iptables targets

```
#!/bin/bash
if [[ $UID != 0 ]]; then
                echo "Please run this script with sudo:"
        echo "sudo $0 $*"
        exit 1
fi

if [ -z "$1" ]; then
        echo usage: $0[pcap-file to create connections] [pcap-file to reuse connections][DNAT ip address] echo
        echo e.g. $0 1M_capture.pcap 10M_merge_capture.pcap 100.66.0.2-100.66.0.254
        echo
        exit
fi
PCAP_CREATE=$1
PCAP_REUSE=$2
IP_ADDRESS=$3

LOGFILE_CREATE="DNAT_create.log"
LOGFILE_REUSE="DNAT_reuse.log"
echo "Removing previous log files"
rm $LOGFILE_CREATE
rm $LOGFILE_REUSE

echo "Flushing all the iptables rules"
echo "....................."
iptables -F
iptables -F -t raw
iptables -F -t nat
iptables -F -t mangle


echo "Marking the DNAT in the PREROUTING chain of iptables..."
```

```
echo "Destination address will be changed to $IP_ADDRESS range."
iptables -t nat -A PREROUTING -p udp -i source0 -j DNAT –to $IP_ADDRESS
echo "Done!!!!!"
sleep 2

for i in {1..10}
do
        echo "*** iteration #$i ***" » $LOGFILE_CREATE
        echo "1. Flushing conntrack"
        conntrack -F
        sleep 2
        echo "2. Creating new connections in conntrack"
        echo "....................."
        taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cache –intf1=source0s $PCAP_CREATE »
$LOGFILE_CREATE
        sleep 2
        echo "*** iteration #$i ***" » $LOGFILE_REUSE
        echo "3. Reusing connections in conntrack"
        taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cache –intf1=source0s $PCAP_REUSE »
$LOGFILE_REUSE
        echo "**************************"
done
```

Script for testing the Marking the packets

```
#!/bin/bash
if [[ $UID != 0 ]]; then
        echo "Please run this script with sudo:"
        echo "sudo $0 $*"
        exit 1
fi
if [ -z "$1" ]; then
        echo usage: $0[pcap-file to create connections] [Mark value]
        echo
        echo e.g. $0 1M_capture.pcap 2
        echo
        exit
fi
PCAP_CREATE=$1
MARK=$2
LOGFILE_CREATE="CONN_MARK_create.log"
echo "Removing previous log files"
rm $LOGFILE_CREATE
echo "Setting the MARK $MARK in iptables...."
iptables -t mangle -A PREROUTING -p udp -j MARK –set-mark $MARK sleep 2
echo "Done!!"
for i in {1..10}
do
        echo "*** iteration #$i ***" » $LOGFILE_CREATE
        echo "1. Flushing conntrack"
        conntrack -F
        sleep 2
        echo "2. Creating new connections in conntrack"
        echo "....................."
        taskset -c 16 tcpreplay –topspeed –loop=1 –enable-file-cach $LOGFILE_CREATE sleep 2
done
```