STATUS OF THESIS

Title of thesis	Genetic Algorithm Optimized Packet Filtering

I OKTA NURIKA

hereby allow my thesis to be placed at the information Resource Center (IRC) of Universiti Teknologi PETRONAS (UTP) with the following conditions:

1. The thesis becomes the property of UTP
2. The IRC of UTP may make copies of the thesis for academic purposes only.
3. This thesis is classified as

☐ Confidential

☒ Non-confidential

If the thesis is confidential, please state the reason:

____

____

____

The contents of the thesis will remain confidential for __________years.

Remarks on disclosure:

____

____

____

[illegible]	Endorsed by [illegible]
Signature of Author	Signature of Supervisor
Permanent address:	Name of Supervisor
Taman Bumi Prima Blok-C1, Bandung, Indonesia.	Dr. Low Tan Jung
Date: 2 May 2013	Date: 2 May 2013

Dr Low Tang Jung
Senior Lecturer
Department of Computer & Information Sciences
Universiti Teknologi PETRONAS

UNIVERSITI TEKNOLOGI PETRONAS

GENETIC ALGORITHM OPTIMIZED PACKET FILTERING

by

OKTA NURIKA

The undersigned certify that they have read, and recommend to the Postgraduate Studies Programme for acceptance this thesis for the fulfillment of the requirements for the degree stated.

Signature:	[illegible]
Main Supervisor:	Dr. Low Tan Jung
Signature:	[illegible]
Head of Department:	Dr. Jafreezal Bin Jaafar
Date:	

Dr. Jafreezal Bin Jaafar
Head
Department of Computer & Information Sciences
Universiti Teknologi PETRONAS
31750 Tronoh
Perak Darul Ridzuan, MALAYSIA

GENETIC ALGORITHM OPTIMIZED PACKET FILTERING

by

OKTA NURIKA

~~A Thesis~~

Submitted to the Postgraduate Studies Programme

as a Requirement for the Degree of

MASTER OF SCIENCE

INFORMATION TECHNOLOGY

UNIVERSITI TEKNOLOGI PETRONAS

BANDAR SRI ISKANDAR

PERAK

APRIL 2013

DECLARATION OF THESIS

Title of thesis	Genetic Algorithm Optimized Packet Filtering

I OKTA NURIKA

hereby declare that the thesis is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTP or other institutions.

	Witnessed by
[illegible]	[illegible]
Signature of Author	Signature of Supervisor
Permanent address:	Name of Supervisor
Taman Bumi Prima Blok-C1, Bandung, Indonesia.	Dr. Low Tan Jung
Date: 2 May 2013	Date: 2 May 2013

ACKNOWLEDGEMENTS

First of all, I would like to thank the Almighty God for the countless blessing towards me so I can finish my studies. My family: my Dad (Muh. Nurhadi Sjukur), my Mom (Farichah Dimyati), and my sisters for their marvelous moral support. I would also thank my supervisors: Dr. Low Tan Jung, Dr. Nordin Zakaria, Dr. Anang Hudaya, and Mr. Abdullah Sani for the precious technical support, inputs, thesis reviews, and motivational words during my research.

I am very thankful for UTP Graduate Assistantship (GA) scheme for the valuable scholarship and the UTP High Performance Computing (HPC) Service Center for the decent facilities and work environment. I thank my UTPian friends and crush for the conducive friendship and the colourful and wonderful school life I have been having here. I cherish every second of togetherness with all of you. Let's keep protecting each other's smile.

ABSTRACT

In our research, we present a method to optimize packet filtering by genetic algorithm. Packet filtering in our work consists of packet capturing and firewall rules reordering. Genetic algorithm is used to automate rules reordering and the discovery of optimal combination of packet capture configuration, in the framework of PF_RING platform and rules ordering. Our method has been tested in different sizes of network traffic load. Genetic Algorithm evolves configuration based on the recorded throughput rates; the higher the throughput the better the solution. Results obtained indicate the effectiveness of the approach.

Keywords: *PF_RING, Packet Capture, Packet Filter, Genetic Algorithm, Optimization, Firewall.*

ABSTRAK

Dalam kajian kami, kami membentangkan kaedah untuk mengoptimumkan penapisan paket oleh algoritma genetik. Penapisan paket dalam kerja-kerja kami terdiri daripada penerimaan paket dan penyusunan semula aturan-aturan firewall. Algoritma genetik digunakan untuk mengautomasikan penyusunan semula aturan-aturan dan penemuan kombinasi optimal konfigurasi penerimaan paket, dalam rangka kerja platform PF_RING dan susunan aturan-aturannya. Kaedah kami telah diuji dalam pelbagai saiz beban trafik jaringan. Algoritma genetik mengembangkan konfigurasi berdasarkan kadar keluaran paket yang direkodkan; kadar keluaran paket yang lebih tinggi menandakan solusi yang lebih baik. Hasil-hasil yang diperolehi menunjukkan keberkesanan pendekatan kami.

Kata Kunci: *PF_RING, Penerimaan Paket, Penapisan Paket, Algoritma Genetik, Pengoptimuman, Firewall.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIDS	Anomaly Intrusion Detection Systems
ALG	Application Level Gateway
ARP	Address Resolution Protocol
BB	Branch-and-bound
BPN	Backpropagation Neural Network
DARPA	Defense Advanced Research Projects Agency
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DNS	Domain Name Server
E-R	Entity Relationship
FPA	Firewall Policy Advisor
GA	Genetic Algorithm
GPU	Graphics Processing Unit
HIDS	Host-based Intrusion Detection System
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Management Protocol
IPS	Intrusion Prevention System
JSS	Job Shop Scheduling
L2TP	Layer 2 Tunneling Protocol
MAC	Media Access Control
MIDS	Misuse Intrusion Detection System
NAPI	New Application Programming Interface
NIDS	Network-based Intrusion Detection System
NPU	Network Process Unit
OS	Operating System
OSI	Open Systems Interconnection
QAP	Quadratic Assignment Problem
RAM	Random Access Memory
RARP	Reverse Address Resolution Protocol
RX	Receive

SA	Security Analysis
SNMP	Simple Network Management Protocol
SPMD	Single Program Multiple Data
TCP/IP	Single Program Multiple Data
TNAPI	Transmission Control Protocol/Internet Protocol
TSP	Traveling Salesman Problem
TX	Transmit
UDP	User Datagram Protocol
URL	Universal Resource Locator
VRP	Vehicle Routing Problem
WWW	World Wide Web

CHAPTER 1

INTRODUCTION

In today's vast utilization of Internet infrastructure, the significance of connection speed, reliability, security, data integrity, and affordability are important for the purposes of using the network efficiently and effectively. Network analysis is often related to the monitoring of network packet transmission quality.

In this research work, network analysis focuses on packet filtering, which consists of packet capturing and firewall rules reordering. Network analysis tools are available from both open source and commercial package. These tools are available as network monitoring tool, network measurement tool, firewall, Intrusion Detection System (IDS), Intrusion Prevention System (IPS), etc. Few of the popular brands are ntop, Cacti, Snort, and Suricata. One of the key performance indicators of these tools is the speed of packet processing. This is one of the research focuses.

The speed of packet capture is indicated by the number of received/captured packets in a specific amount of time. Different packet capturing techniques might yield different results. Majority of researches in packet capturing attempted to make modifications to existing technologies, in order to improve the capturing speed. A part of this research work is to optimize the packet capturing rates, by utilizing an existing kernel patch called PF_RING. PF_RING kernel parameters are evolved using genetic algorithm to find the optimum setting for different levels of traffic load.

Another feature of PF_RING is the specification of firewall rules, which is the second focus of optimization in this research. In this research, firewall rules reordering optimization is suitable in the cases where there are more than one rules with 'accept' action, than the ones with 'drop' action, especially when the new network administrator is faced with existing rule sets and new rules are needed for insertion. Therefore reordering is highly needed. This research work puts both packet capture kernel parameters optimization and firewall rules reordering in the same methodology framework.

1.1. Problem Description

The performance of packet filtering is influenced by hardware and software factors. Different packet capture technologies utilized hardware in different ways which generate various throughput rates. Packet capture rate is also "adjustable", which means that its configuration setting determines its optimal throughput. The configuration consists of parameters, that if optimized, will generate better throughput rate.

Each level of input traffic load requires different packet capture treatment. For example, capturing network traffic with small size packet (64 bytes) needs a setting which is not the same as for capturing traffic packet with big size (1500 bytes). Our research utilizes genetic algorithm to find the near optimum combination of setting the parameters for distinct network traffic streams, starting from the stream with 64 bytes upto 1536 bytes of packet size. The optimal setting is important, especially in a case where the size of packet for transmission can be predicted, or is indicated earlier by the sender.

A new network administrator who is to insert new rules into an existing firewall rule set, must reorder the rules so that the rules to accept packets are placed before the rules to drop the packets. Thus, those rules are processed with higher priority, to eventually accelerate packet capturing performance. In a condition where the number of rules is high, automation of rules reordering will speed up the reordering process. Besides optimizing packet capture parameters setting, this research also tries to tackle this problem by using genetic algorithm to reorder the rules automatically. Both processes are done simultaneously in one genetic algorithm.

1.2. Objectives

This research aims to achieve the following objectives:

1. Design a framework for integrated packet capture and firewall rules reordering optimization. The current firewall rules reordering methodologies do not collaborate packet capturing kernel parameters optimization to accelerate the overall packet filtering (packet capturing and firewall rules reordering) process. This research gap is a part of our objectives.

2. Integrate packet capture parameters and firewall rule set into genetic algorithm process. The simplicity of our proposed genetic algorithm methodology will cover both packet capture kernel parameters and firewall rule set in one genetic algorithm application, therefore two optimizations are run simultaneously.

3. Generate optimized combination of packet capture kernel parameters for different levels of traffic load, based on maximum throughput rate. Different input traffic loads require different settings of kernel parameters. The genetic algorithm application will select the optimal combination of kernel parameters for each input traffic load based on the recorded fitness value, which is the throughput rate.

4. Generate optimized order of firewall rule set, based on maximum throughput rate. A set of firewall rule in random orders is evolved in a genetic algorithm application, with the purpose to reorder the rules so that the rules with 'accept' action are positioned at the front row of the rule set. Therefore they are processed faster and the network speed will be optimal.

1.3. Thesis Outline

This thesis is written for readers with no or entry knowledge of computer network theories. Chapter 1 introduces the area of research focus and contributions. Chapter 2 informs about the basic knowledge of computer networks and previous works related to this research focus area. Later chapters discuss about methodology, results and analysis, conclusions, and potential future works. Our thesis consists of the following chapters:

Chapter 1: Introduction, Problem Description, Objectives, Thesis Outline

Chapter 2: Literature Review, The TCP/IP Stack, Intrusions into the Network, Deployment of Firewall for Protection, The Previous Works to Precise the Firewall Rules, The Assisted Firewall Implementations, Introduction to Packet Capturing, Current Packet Capturing Technologies, High Speed Packet Capturing and Packet Filtering, Genetic Algorithm (GA) in Packet Filtering System, Network Penetration Testing

Chapter 3: Methodology, Introduction to Genetic Algorithm Library, Packet Capture Platform to be optimized by Genetic Algorithm, PF_RING Parameters to be optimized by Genetic Algorithm, PF_RING Application's Representation of Chromosome

Chapter 4: Experiments Results and Analysis, Experiment Software and Hardware Environment, Experiment Results and Discussion

Chapter 5:, Conclusions, Contributions, and Future Work

CHAPTER 2

LITERATURE REVIEW

2.1. TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) stack is a simplification of the seven (7) OSI (Open Systems Interconnection) layer communication standard. However, historically speaking, TCP/IP preceded OSI seven layer in data transmission protocol development as told by Held [1]. TCP/IP stack is also called *Internet Protocol Suite*. While the seven (7) OSI layers were defined by ISO (International Organization for Standardization), TCP/IP was created by US DARPA (Defense Advanced Research Projects Agency). TCP/IP is a communication standard that manages how the data is exchanged from the sender to the receiver regardless the sender/receiver's differences in different brands, or even different machine types (desktop computer, laptop, mobile phone, tablet, etc).

The TCP/IP standard was built in stacks, for it to be easier for the communication experts to improve, or add new modules or technologies into the TCP/IP system. Furthermore, it is easier for learners to understand how data is exchanged by looking at the layer per layer basis instead of a one big chunk of communication module.

When talk about data journey or data exchange, it is usually referred to the seven OSI layer or the TCP/IP stack. Generally it is fine because both standards consist of the same protocols or technologies. The difference is that one categorizes the protocols into seven (7) layers and the other classifies protocols into four (4) layers. In fact there are other standards that define communication protocols into different number of layers. For example Arpanet Reference Model 1982 (RFC 871) that classifies protocols into three layers or the Juniper Networks Learning Academy that uses five layers of data communication stack. Practically, among the network engineers, the TCP/IP and the five layers communication stack are the most popular ones, but since the official standard is TCP/IP, so TCP/IP is picked as a standard for this thesis.

As described by Panwar et al. [2], TCP/IP stack consists of four layers, which are (from bottom to top): Data Link layer, Network layer, Transport layer, and Application layer as shown in Figure 2.1.

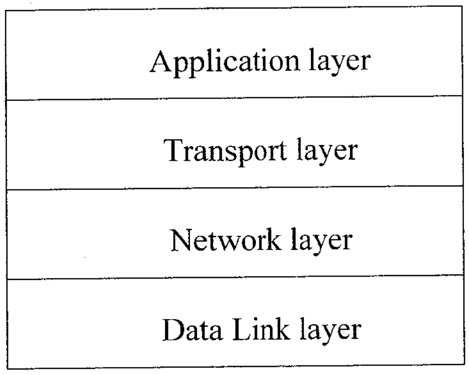


Figure 2.1. TCP/IP Stack

Each of the layers serves each other back and forth. First of all, the discussion is on what occurs within every layer as according to Panwar et al. [2], Held [1], Lammle and Barkl [3], and Coll [4], [5], [6].

a. **Application layer**. This layer provides interface to bridge user application and the formatting it needs to be forwarded by the immediate underlying layer (Transport layer). It manages protocols for sender/receiver application communication and regulates end user interface specifications regardless of hardware platform, operating system, and other different features of sender and receiver.

   Some Application layer protocols include: HTTP (Hypertext Transfer Protocol) that provides WWW (World Wide Web) service, Telnet that is used to connect remotely to a computer host, DNS (Domain Name System) that translates domain name to IP (Internet Protocol) addresses, SNMP (Simple Network Management Protocol) that manages network entities remotely and centrally, and DHCP (Dynamic Host Configuration Protocol) that automates network addressing configuration for client computers that connect to a gateway. Most of these protocols are for client/server application. The client shall hold the data that need to be transferred.

b. **Transport layer**. It provides data transport service for the Application layer. It manages the protocols to deliver the level of transmission quality whether reliable (connection oriented with error checking and delivery acknowledgement) or unreliable data transfer (connectionless oriented without error checking and no delivery acknowledgement). Importantly, it covers the upper layer (Application) from the complexities of the network. Two main protocols at this layer are:

1) **TCP (Transmission Control Protocol)**: TCP provides reliable and connection-oriented transmission of data over the IP (Internet Protocol) network. It fragments large data from application into segments, then it gives numbers to each segment so that they are in sequence. Hence, the destination node can reassemble the segments into proper order. The receiver will send acknowledgement for every segment accepted. The unacknowledged segments will be resent by the sender that keeps a timer for acknowledgement of each segment.

   TCP will establish connection first before sending process begins. The sender's TCP will contact the receiver's TCP to establish a virtual circuit which is why TCP is known as connection oriented protocol. Then both parties shall agree on certain parameters such as the amount of information that will be sent before the receiver sends an acknowledgement. Once the TCP session is established, the Application layer can use the established session to transmit information. It should be mentioned here that additional packet header information that TCP adds to every segment that would take up additional bandwidth, so these information should be used only in transmission where accuracy and reliability are highly desirable.

2) **UDP (User Datagram Protocol)**: It is a connectionless transport protocol hence it does not warrant proper and accurate transmission. On the other hand, for its least-featured and simplicity, it transmits information much

faster than TCP, making it suitable for speed-conscious applications like video or audio streaming. It does have few similarities like TCP, for instance it may break information into segments and gives them number, but it does not sequence the segments and does not give attention in what order the segments arrive at the recipient, and it does not even support acknowledgement of segment arrival.

Furthermore, UDP does not build or establish virtual circuit before sending information. It just sends the information without wanting to know whether the receiver is ready to accept the information or not. Therefore, it is considered as connectionless transport protocol.

c. **Network layer**. This layer provides routing service to each packet in order to deliver it to the final destination by assigning each host an IP address. It is also the most talked about layer among the network engineers since it takes advantage of some popular and most used routing protocols e.g. RIP (Routing Information Protocol, OSPF (Open Shortest Path First), and IS-IS (Intermediate System to Intermediate System). IP is the main protocol within this layer, while the other protocols exist to support it. Below discussion involves some most popular protocols within the Network layer.

1) **IP (Internet Protocol)**: IP inspects packet header's IP address to decide the best path or which next node it should send packet to based on the information in the routing table. To accomplish its job, IP needs to know which network nodes are on by looking at these nodes' IP addresses (logical) and what their IDs are on each of their networks by inspecting the physical addresses or hardware addresses or the MAC (Media Access Control) addresses. These MAC addresses are actually their real addresses since it is unique for each device.

IP accepts segments from the Transport layer and then fragments them into datagrams or packets. Each datagram is assigned an IP address of the source/sender and the destination node. There are two (2) pairs of IP addresses that each datagram will have. First pair is the network address which consists of the original sender and destination node's IP address. This pair of addresses will not be altered along the route by the transit nodes and it will be checked with the routing table to determine the next best path to reach the destination. The second pair of addresses is called the link address which consists of the previous source hop or path's IP address and the next hop or the best path's IP address that is determined by the transit node based on its routing table. This pair of link addresses is always altered by the transit node until the segment reaches its final destination.

Therefore, the journey of datagram or packet is described as mentioned below:

- The sender sends datagram to the best transit node based on routing table.

- The transit node checks the pair of network address to inspect the destination address and makes routing decision and sticks an additional pair of address called the link address that consists of source address which is its own IP address and the next hop address which is the IP address of the next transit node.

- The next transit node that receives this datagram will inspect the network address then makes routing decision and then alters the pair of link address, so now the link address consists of this current transit node's IP address as the source address and the next hop's IP address which is derived from its routing table as the destination address.

This process of the pair of link address alteration will goes on until the datagram reaches its final destination, which is the destination IP address that is labeled on its pair of network address. On the receiving side, IP reassembles the datagrams received back into segments in an orderly manner.

2) **ARP (Address Resolution Protocol)**: It is a protocol used to resolve computer's physical/MAC address (real address) based on its assigned IP address (logical).

   When IP is about to send a datagram, it has been informed about the destination's IP address. However the lower layer which is Data Link does not look at IP address. Instead it looks at the real address (physical/MAC address) of the destination. Therefore IP needs to add the destination's MAC address too. IP utilizes ARP to do this. ARP will broadcast request to inquire about which node has specific IP address, and the respective node will reply by telling its MAC address. Information about which node has which IP and MAC address is stored in ARP table.

3) **RARP (Reverse Address Resolution Protocol)**: RARP is activated when a destination has no capability to initially know its IP address. Therefore a RARP request containing the destination's MAC address will be broadcasted, so that a chosen RARP server will reply with an information about what IP address is assigned to that destination node.

4) **ICMP (Internet Control Management Protocol)**: This protocol is the most commonly used protocol to test network reachability. A popular utility that makes use of ICMP packets is 'ping'. Ping transmits a series of echo messages to a destination node address. If the host is reachable and permits ICMP reply then it shall reply with a series of echo reply messages, the round-trip time it takes for every echo message sent and the arrival of its echo reply will be computed by the sending node that sets a timer. In other words, the round-trip time can be said as the delay.

   Finally, ICMP provides facility to control communication message and report the errors. ICMP is used by network entities (switches, routers, firewalls) and computer nodes to test the network or send datagram problems back to the datagram originator.

d. **Data Link layer**. Data Link layer manages hardware/MAC addressing and provides protocols for the physical transmission of the data. Main processes that occur within this layer are:

1) Convert IP datagram into a series of bits for physical transmission. This series of bits make up a frame.

2) Specify the physical/MAC address.

3) Make sure that the series of bits of a frame has been properly received by the destination node by calculating CRC (Cyclic Redundancy Checksum).

4) Specifying the Data Link technology, for example it can be Ethernet (FCFS-based; First Come First Served), Token Ring (the node that owns the token will send first), and etc.

5) Specifying the physical attributes such as the media (cable or wireless), connectors (RJ-45, RJ-11, etc), electrical signaling, timing rule, and so on.

2.2. Intrusions into the Network

The network operation could be disturbed by the following items according to Canavan [7] and Vacca [8]:

1) **Threats**. A threat is "anything that can disrupt the operations, functions, integrity, or availability of a network or system". It can exist in various forms including an act of nature. Aissa et al. [9] further classifies threat into three types that are threats on communication protocols, threats on operating systems, and threats on the information.

2) **Vulnerabilities**. A vulnerability is "an inherent weakness in the design, configuration, implementation, or management of a network or system that renders it susceptible to a threat".

3) **Intrusions/Attacks**. An intrusion is "an unauthorized penetration of a computer in your enterprise or an address in your assigned domain". There are loads of types of attacks, but generally it is classified into two general categories of passive or active attack. Passive attack mainly deals with monitoring and recording activity such as packet sniffing and traffic analysis; a popular tool for this is Wireshark (previously named Ethereal). While active attack involves dynamic activities to bring down the network e.g. Denial of Service and brute force attack.

A weakly configured network could be exposed to unauthorized access or intrusions. The known types of intrusion per each TCP/IP stack layer plus Physical layer as consolidated from Canavan [7], Dunsmore et al. [10], Toxen [11], Maiwald [12], Garfinkel [13], Lockhart [14], and Gheorghe [15] are:

a. Physical and Data Link Layer Intrusions:

The Physical and Data Link layer can be breached by one of the following methods.

1) **Physical**: gaining access by taking opportunity when the workstation is unmanned and logged-on or by stealing the workstation. Physical attack can also be in the form of cable cuts, high voltage applied on copper cables, wireless links jamming, electromagnetic brought near copper cables

2) **Social Engineering**: a social trick to bring a person who holds some information about a network to tell the information without any concern that it might be dangerous.

3) **Eavesdropping**: an attempt to listen to a conversation between two parties by tapping the communication line.

4) **Snooping**: a process of searching useful information by going through papers or text files on a computer.

5) **MAC Flooding**: flooding a MAC table of a switch with invalid source MAC addresses hence the switch will broadcast incoming packets out of all ports.

6) **DHCP Attack**: an attacker accomplish this by broadcasting numerous DHCP requests using falsified MAC addresses, so the DHCP server will have no more IP addresses to allocate to the legitimate users.

7) **ARP Attack**: a typical Denial of Service attack by broadcasting ARP messages with falsified MAC or IP addresses.

8) **VLAN (Virtual Local Area Network) Hopping**: attacker utilizes the VLAN trunking feature that is enabled by default on some switches, therefore the attacker can tag their packet with the target's VLAN ID to create communication.

9) **Network Loop**: Network loop easily occurs by connecting two ports on the same switch with the same VLAN ID using a cross cable, this loop brings infinite packet journey within that switch that could crash the processor chip.

10) **STP (Spanning Tree Protocol) Manipulation**: attacker broadcasts topology change BPDU (Bridge Protocol Data Units) to force STP topology to be recalculated and since the recalculation takes around half minute hence it could bring up Denial of Service.

b. Network Layer Intrusions:

The invasions via Network layer can be done in several ways as mentioned below.

1) **Scanning**: a process of discovering a network topology by sending request message such as ping, traceroute, or netstat.

2) **Denial of Service**: an activity to overwhelm the processing capacity of a networked computer or server, hence the services it provides will halt.

3) **Distributed Denial of Service**: a simultaneous attack from a group of computer hosts to overwhelm the target host.

4) **Sniffing/Man in the Middle**: an action to monitor and record information flowing between the sender and receiver.

5) **IP Address Spoofing**: an act to gain access to the network by using permitted IP address so that the attacker appears as a legitimate client.

6) **Ping of Death**: a typical Denial of Service attack which the attacker sends huge ICMP echo message to overwhelm the target machine.

7) **Smurf Attack**: attacker sends an ICMP echo request to the broadcast address of a network with the return address of a target host, all computers inside that network group will flood the innocent target host with ICMP echo reply messages (this requires IP reachability from that network group to the target host).

8) **Routing Protocol Attack**: it utilizes the misconfiguration of dynamic routing protocols of a router to inject false routes in order to create a Denial of Service.

9) **Teardrop Attack**: the attacker sends fragmented IP packets with their offset values altered so that the target host cannot reassemble the packets which eventually causes kernel panic in Linux or blue screen in Windows.

c. Transport Layer Intrusions:

The exposures of Transport layer by an intruder might affect the services and privacy of the data. The below methods are possibilities to evade the network via Transport layer.

1) **Session Hijacking/Man in The Middle/Interception**: an act of taking over a connection session, usually by gaining access to the gateway router and it mostly involves IP spoofing too.

2) **Sequence Number Spoofing**: an unencrypted TCP/IP network connection can be hijacked by an attacker who includes himself into the connection by using the next expected sequence number. The sequence numbers are used to maintain the connection.

3) **Replay Attack**: an interception and storage of a transmission between sender and receiver then retransmit it later. This can lead to Denial of Service.

4) **SYN Flooding**: another kind of Denial of Service attack where the attacker sends TCP establishment request (SYN message) to the target host but with invalid return IP address so that the SYN ACK message from the target host will reach nowhere. Numerous SYN messages with invalid return IP address could crash the target host.

5) **UDP Flooding**: sending loads of UDP packets to any ports that shall crash the target host because the host will keep determining which UDP port addresses which application.

6) **Port Scanning**: a process of finding which ports are opened, therefore running applications could be discovered and exploited.

d. Application Layer Intrusions:

The easiest intrusions to do are done through the Application layer. Many softwares are available to perform mischief at this level. The followings are some of the common techniques.

1) **DNS (Domain Name Server) Poisoning**: a mischief to alter the DNS table contents so it will direct the URL (Universal Resource Locator) name to a false or malicious website's IP address.

2) **Password Cracking**: a procedure to find a password by comparing a dictionary-based password file where each word is encrypted with a stolen official password file that contains legitimate password words that are encrypted with the same algorithm as the attacker's.

3) **War Dialing**: a series of action to dial a range of a company's phone extension numbers and once it hits a modem, the modem log will be learned and the attacker could gain access by taking over the computer that is connected to the modem.

4) **Spamming**: Flooding a target's email server by sending numerous emails that might contain falsified return address, hence the email server and the return address could crash or freeze.

5) **Buffer Overflow**: loading a malicious code or huge amount of data into the memory buffer of an application.

6) **Server Vulnerability Attack**: exploitation on a flaw or an undocumented command in a server to gain access.

7) **Web Server**: attacker attempts to harm the web server by trying to upload malicious scripts into the web server.

The above described intrusions can be blocked by deploying a module called a firewall which can be implemented in the form of software or hardware. Firewall is discussed in the next section.

2.3. Deployment of Firewall for Protection

Firewall is a form of protection that allows a local network to connect to external network with certain level of security policies as told by Zwicky et al. [16]. Bernstein [17] includes firewall as a part of IT (Information Technology) Management Infrastructure. The firewall is placed as a separator between the internal and external network (or the Internet) as shown in Figure 2.2.

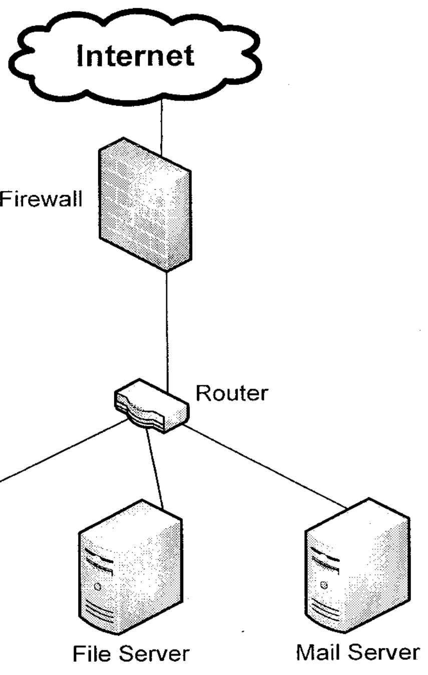


Figure 2.2 The Positioning of a Firewall

Firewall is interpreted differently according to the way it filters the packets within a network. Firewall can be in the form of software-based packet filtering, route-filtering, proxy server, stateless packet filtering, stateful packet filtering, and even IDS (Intrusion Detection and Prevention Systems). Therefore, firewall can inspect packets from Network layer up to Application layer.

Generally, firewall serves to achieve some purposes as mentioned below by Zwicky et al. [16]:

1. To prevent people to gain access to certain nodes or subnets.
2. To prevent people to reach other area of network defense.
3. To prevent people to leave the network at a controlled point of network.
4. To perform NAT (Network Address Translation).

Types of firewall that generally known by network engineers and described by Canavan [7], Ziegler [18], and Ogletree [19] are:

a. **Static/Stateless Packet-Filtering Firewall**: a packet filtering process based on simple if-then rules. Generally it filter packets based on Network layer information such as source address, destination address, source port, and destination port. The respective firewall rules do not change over time unless firewall is reconfigured.

b. **Dynamic/Stateful Packet-Filtering Firewall**: this kind of firewall maintains the TCP connection and is informed about every packet type (SYN, ACK, etc) and can terminate the connection if unexpected packet type exists. It is called dynamic because the firewall can react to abnormal message type.

c. **ALG (Application Level Gateway)**: ALG stands in the middle between the client and the server and it knows what application that it mediates. It inspects the Application layer's payload to ensure data integrity.

A research by Govaerts et al. [20] realized the highly increasing need of firewall as there were numerous reported network security incidents. They proposed a new technique of setting the appropriate rules in firewall packet filtering, called formal logic that based on Entity Relationship (E-R) between network packet attributes such as source address (s_a), destination address (d_a), destination port (d_p), protocol (p), and action (a) that was either accept or deny. The formulation of this firewall ruling was: rule(a, s_a, d_a, d_p, p). This formulation would then be implemented using Linux-based packet filter software IPChains and Cisco firewall device named Cisco PIX.

Govaerts et al. [20] also suggested that the default action or the action that should be applied for the undefined packet in the rule was deny because this would reduce the complexity of the firewall rule and decrease the security holes. Their formal logic approach managed to tackle packet matching problems such as shadowing, correlation, generalization, redundancy, and irrelevance. These problems were also recognized by Alfaro et al. [21].

However, the firewall implementation approach by Govaerts et al. [20] requires new installation of firewall device either using a specific computer installed with IPChains or a dedicated proprietary firewall which in this case is Cisco PIX. Moreover, Govaerts et al. [20] did not proceed into equipping the firewall with IDS feature which is highly comprehensive to detect latest kinds of intrusions depending on the IDS attack database content.

2.4. The Previous Works to Precise the Firewall Rules

The body of firewall rules generally contains a set of if-then construct which matches packets based on source/destination address, source/destination port, and protocol type with action applied as either accept or deny. This set of firewall rules eventually has a default action (accept or deny) that is defined at the very last rule line. Each firewall interface represents a Local Area Network (LAN). The interfaces that have similar security requirements are grouped in the same security zone. Eventually, each set of firewall rule is applied within every interzone firewalling rule.

Anomalies may occur if a packet that is meant to match a rule also matches the previous rule hence it creates a conflict. If a rule does not match any packet or so called useless, and if the source and destination address belongs to the same security zone, then the packet will not go through a firewall.

Alfaro et al. [21] proposed a method to eliminate above anomalies by creating intra-component algorithms that allows administrators to rewrite firewall rules by generating a firewall rule set that contains only deny actions if the default rule is accept or generating a firewall rule set that contains only accept action if the default rule is deny. Therefore we can bring up two hypotheses related to these patterns.

The first hypothesis is to allow packets from known good source and dropping the others and the second is dropping packets from known bad source and allowing the others. This research by Alfaro [19] et al. did not implement the resulted firewall rule set. They only created and optimized the firewall rule set without implementing or testing it within a firewall. A simulation or emulation could have been done to test the firewall rule set against data packet traffic.

Al-Shaer and Hamed [22] made a comprehensive research to clean-up firewall rule-sets that contain anomalies for both inter-firewall and intra-firewall rule sets. Each rule-set row consists of protocol type, source IP address, source port number, destination IP address, destination port number, and action. Every row is destined to take action on distinct type of traffic. The anomalies for intra-firewall rule-sets are described below:

1. **Shadowing anomaly**: This happens when a preceding rule matches the respective rule, which in this case the respective rule will never be triggered.
2. **Correlation anomaly**: This occurs when some packets that are supposed to match the first rule, also match the second rule and vice versa, but the first rule and the second rule have different actions.
3. **Generalization anomaly**: A respective rule is a generalization of another rule if it specifies a more general subnet but with different filtering action.
4. **Redundancy anomaly**: A rule is a redundant to another rule if it gives the same treatment on the same packets with another rule. Even the firewall policy will not be affected if the redundant rule is removed. Usually the redundant rule is in the subset form.
5. **Irrelevance anomaly**: A rule is irrelevant if the source and destination address of the packet does not match any reachable network subnet of the firewall.

While the anomalies for inter-firewall rule-sets are similar but with different perceptions and some distinct anomalies are:

1. **Shadowing anomaly**: This is when a packet is denied by the upstream firewall, but actually accepted by the downstream firewall.
2. **Spuriousness anomaly**: It happens when a packet is accepted by the upstream firewall, but later denied by the downstream firewall.
3. **Redundancy anomaly**: A redundant rule might exist in the downstream firewall if it rejects a packet which is already denied by the upstream firewall.
4. **Correlation anomaly**: If some packets meant to match the upstream firewall also match the downstream firewall, or vice versa but with different filtering action, then the rules in the upstream and the downstream firewall are correlated.

The above anomalies can be detected by a tool created by Al-Shaer and Hamed [22] called "Firewall Policy Advisor" or FPA in brief. This Java-based software implements Anomaly Discovery Algorithm for both intra-firewall and inter-firewall rule-sets, which can detect the previously mentioned anomalies and warn the administrator who will then decide the action to be taken based on his discretion. The rule of thumb is that a rule that is a subset of another rule must have smaller order number, or in other words, it goes in the rule set before its superset pair.

For intra-firewall anomaly discovery process, the algorithm compares each field in rule R_y (subsequent rule after R_x), sequentially protocol type, source address and source port, and then destination address and destination port. If found that a field of R_y is within the subnet range of R_x with the same action handling, then R_y is redundant to R_x. Or if the action of R_y differs from R_x then it means R_y is covered or deactivated by R_x. Reversely, if a field of R_y is the superset or equals to a field of R_x with same action handling, then it means R_x is the one redundant to R_y. If the action is different, then R_y is the general form of R_x.

Another case is when several fields of R_x equal or are subsets of R_y, and the rest of the fields of R_x are the superset of R_y but the action of R_x and R_y are different, then it means R_x and R_y are correlated which is another anomaly that must be tackled. Basically, every rule must be guaranteed to be disjoint to each other. The anomaly discovery algorithm can be found in a paper by Al-Shaer and Hamed [81].

In an environment where multiple connected firewalls exist, the rules comparisons are represented by R_u (a rule in the upstream firewall) and R_d (a rule in the downstream firewall). Sequentially in the same manner as in intra-firewall anomaly discovery algorithm, every field of R_d is compared to its corresponding field of R_u. If R_u inclusively matches R_d with the same action which is "accept", then it means R_d is shadowed or covered by R_u. Or if the action of R_d is "deny" then upstream will keep forwarding packets that R_d will eventually drop, which in this case it is a spurious traffic towards R_d.

The inter-firewall anomaly discovery process is executed among the firewalls that are connected to separate domains which have connectivity to each other. First in the process, all firewall rule sets that fence separate domains are aggregated. Then the algorithm will capture the network routes between domains as an input, and determine which firewalls stand-in between each path. And then, each firewall is cleaned from intra or inner anomalies, by running the intra-firewall anomaly discovery algorithm.

After every firewall is cleaned from inner anomalies, for each path (one source, one destination), the policy tree is created for the most upstream firewall (nearest to the source domain). After that, the rules from the other following firewalls are added to this policy tree, and the rules that match the respective path are taken into account. Finally, after every path is evaluated, all resulting rules that might create anomalies are reported. While the unselected rules are reported as irrelevant as they do not belong to any path in any domain. Based on these reports, the network administrator will decide which rules should be rewritten or even removed. Al-Shaer and Hamed [23] details the algorithm to build the policy tree.

The test-bed itself for intra-firewall anomaly discovery consisted of four rule sets to be compared to each other. Set 1 was a set of rules that were distinct in destination address. Set 2 builds up rules with distinct source addresses. Set 3 describes each rule as a superset match of the preceding rule (worst case), and lastly the set 4 that consisted of the random mixture of rules from set 1, 2, and 3. On each set, Al-Shaer and Hamed [22] gave variations of rule amount ranging from 10 – 90 rules each set.

The fastest computation time for this anomaly discovery recorded was for set 1, the second fastest time was captured for set 2. Set 3 had the longest computation time, while set 4 had longer processing time than set 2 but were faster to process than set 3. These results are considered reasonable, logical, and acceptable. Even for the worst case, the computation time ranging from 20 ms to 240 ms (10 – 90 rules). Additionally, the approximate increase in time for every one rule addition is around 2.1 – 2.8 ms. This amount is an acceptable load for the processor.

The second test-bed for the inter-firewall anomaly discovery process involved distributed firewalls across four networks, with the following hierarchy for each network: (1) 2-2-2, (2) 3-2-2, (3) 3-3-2, (4) 4-3-3. The first number indicates the amount of branches that the root or top node has. The second number represents the amount of branches that each node of level 2 has. The third number shows the amount of branches that every node of level 3 has.

The FPA software then took input of the above hierarchical topology and was run for every network with variations of the number of rules ranging from 10 – 50 rules for every single firewall. The resulted computation times were linear to the rule complexity and the amount of domains (more domains bring up more network routes). The more complex and or more domains involved then the longer its computation time. In details, for network 1 (8 domains) and network 2 (12 domains), the computation times ranged from 3 – 40 seconds. While for network 3 (18 domains) and network 4 (27 domains), the computation times ranged from 11 – 180 seconds.

The use of FPA software implemented with anomaly discovery algorithms had been proven to be an influential assistant, specifically to optimize the on-field accuracy of the network security policy. However, there are other things that influence the performance of the network packet filtering, such as the hardware specifications and the speed of the in-kernel packet capturing software module itself.

The hardware capabilities can be fully utilized if the kernel module knows how to use the hardware in the best way possible. In this case there are some parameters that can be adjusted. For example the memory assignment management, and how the incoming packets are queued and processed.

Our research tries to simplify the approach, by optimizing the accuracy of the firewall rules and the kernel module parameters at the same time. Genetic algorithm is chosen to be the research element, because of its generic and extendable properties, making it applicable for different permutation cases. In this research, genetic algorithm is applicable to both firewall rules and packet capturing optimization.

2.5. The Assisted Firewall Implementations

Original firewall does not detect the payload of the packet as told by Sourour et al. [24]. The thing is that even though the packet comes from permitted source address and source port, it may contain malicious codes. In other words, attacks may come from Intra network which is actually the major source of attack found by Sourour et al. [24].

Therefore, Sourour et al. [24] proposed a heterogeneous network security infrastructure that included IPS (Intrusion Preventon System), VS (Vulnerability Scanner), and honeypot along with firewall to strengthen the packet filtering infrastructure.

IPS is an advancement of IDS, where IDS detects intrusion by looking at the packet's payload and sends an alert to the administrator, but does not terminate the attack progress. The termination of malicious network connection can be done by IPS according to Sourour et al. [24]. Sourour et al. [24] proposed IPS implementation by using Snort-inline in peering mode (active-active) that consisted of six IPS servers. So if one server is updating its attack database and shall reboot to finish the database update, then the other IPS servers will take over the filtering load.

Sourour et al. [24] tried two combinations of security infrastructures. First was the collaboration of honeypot and firewall. The second was the collaboration of honeypot and IPS. Honeypot is a flexible network security tool that is placed in the network as a decoy to attract attackers and monitor the attack processes or techniques as described by Prathapani et al. [25], where the attack characteristics can be learned.

In honeypot-firewall collaboration, the firewall modified its rule set depending on the new attack information provided by the honeypot, while in honeypot-IPS collaboration, the IPS servers would filter three types of packets that were identified as legitimate, intrusive, and unknown. The legitimate would be accepted, the intrusive would be dropped, and the unknown ones would be forwarded to honeypot for further analysis for further action.

Sourour et al. [24] succeeded in building a dynamic security infrastructure that can adapt to new attacks by utilizing honeypot. Unfortunately they did not integrate firewall and IPS. Furthermore, their proposed IPS implementation required new installation of new hardware/computers to be placed in front of the network, and they also did not discuss about the implementation of heterogeneous security system within the existing network that had routers before the Internet. Additionally, Sourour et al. [24] could have deployed comprehensive traffic penetration using various types of packet injection, instead of only relying on ping packet injection to test their security framework.

Cai et al. [26] identifies two types of IDS which are MIDS (Misuse Intrusion Detection System) and AIDS (Anomaly Intrusion Detection Systems) based on the detection technique. MIDS detect the pattern of known attacks while AIDS detects intrusion based on calculating the standard deviation between the current packet and the known normal patterns.

Sheikhan et al. [27] classifies AIDS and MIDS into three classes which are statistical-based, knowledge-based, and machine learning.

Another IDS classification is made by Jin et al. [28] that groups IDS into two types which are Host-based IDS (HIDS) and Network-based IDS (NIDS) according to the source of detected data. HIDS works by recording internal data such as the integrity of file systems, audit logs, network events, and the sequence of system calls. On the other hand, NIDS operates by analyzing captured network traffic to detect intrusions. The most popular NIDS is Snort which is an open source light NIDS that utilizes Misuse-based Intrusion Detection or MIDS according to Zulkernine et al. [29].

Zulkernine et al. [29] modified Snort so that it could contain context information in its `msg` option, this was helpful to tackle more complicated and multi step attack scenarios. For example it would not rise alert for an attack that was not meant for the respective operating system. However this approach has not been tested in firewall in an enterprise telecommunication network environment.

Folino et al. [30] informed a new kind of NIDS called Distributed Intrusion Detection System (dIDS) which is a group of IDS installed over a large geographically separated network that communicates with each other to monitor the network in enhanced way, analyze the security incidents, and provide quick attack database.

In a distributed environment where IDSs are placed in several subnets or points of the networks, the centralization of IDS management becomes an issue since it will affect the time to finish configuring or updating the IDS software. This issue was studied by Jin et al. [28] that came with a VMM-based IDS that utilized a Virtual Machine Monitor (VMM) using VMFence concept that monitored all the VM instances that were installed in the privileged VM of the enterprise cloud network. Therefore packets that go through each VM instance could be monitored and filtered by the single monitoring IDS VM instance, this was made possible because every VM instance communicated with each other via a virtual bridge.

The above centralized IDS management has inspired researchers to implement a central management IDS for the physically distributed IDSs in a distributed enterprise network environment with multiple external interfaces that are vulnerable to attacks. The management of distributed IDSs is highly important to speed-up the configuration of each IDS software so that the duration of vulnerability can be reduced.

Naveed et al. [31] introduced a hybrid approach to create and update access list on the existing network device, in this case, a Cisco 2691 router. A Linux Fedora computer connected to the Cisco router was installed with Snort as an IDS application and Snort would log the alert to the MySQL database immediately after it recognized an intrusion. A web-based application was built to automatically query the attack alert everytime new entry came into the MySQL database and then it would use the alert to generate Cisco Access List new rule set to be executed on the connected Cisco router.

The above approach by Naveed et al. [31] was quite affordable for the enterprise company, but somehow it needs additional budget to setup the computer that acts as an IDS sensor. The activation of new access list rule set goes through two phases. First is the intrusion detection and database insertion at the IDS sensor computer, and second is the access list insertion at the Cisco router. This is not appropriate if the router acts as the gateway to the external network especially the Internet because the traffic will go straightly into the router. Hence the router will have more detection concerns than the IDS sensor computer. The efficiency of this two-host intrusion prevention cycle could be further accelerated, if the Snort IDS is integrated with PF_RING kernel module, which will be discussed in the later section.

Regarding the detection techniques implemented by Naveed et al. [31] , the packet filtering process only blocks the packets coming from known malicious IP addresses. This becomes a drawback if the attacker uses dynamic IP address. Therefore the database update and its transfer to the connected router will happen frequently which might create vulnerable time slots. The lack of port number filtering within the access list testing conducted by Naveed et al. [31] made the true performance speed of this framework undiscovered, therefore deeper testing regarding access list completeness and its performance needs to be assessed.

2.6. Introduction to Packet Capturing

Packet capture is a method that enables computer to capture and examine the contents of the data packets that go into it through network card (SonicWALL [32]). These packets consist of data and addressing or routing information as mentioned below by SonicWALL [32]:

- Interface ID
- MAC address
- Ethernet type
- Internet Protocol (IP) type of service
- Source IP address
- Destination IP address
- Source port number
- Destination port number
- Layer 2 Tunneling Protocol (L2TP) payload details
- Point to Point Protocol (PPP) negotiations details

Packet capturing is useful to know what is happening in the network and to secure and assist in network troubleshooting. A popular network sniffing tool called Wireshark details the purposes of packet capturing in their official website (Wireshark Purposes [33]) as follows:

- To troubleshoot network problems
- To examine network security issues
- To debug protocols deployments
- To study the mechanism of network protocols

The captured packets from the kernel can be forwarded to application layer space for further processing depending on the application type that uses them. For example is the integration of kernel packet capture acceleration module named PF_RING with a popular Intrusion Detection Systems (IDS) software called Snort. However, the Snort is modified so that it works in inline mode, which makes it capable to modify or drop packets in real time when the respective packets match a rule of denial. Inline Snort can also send reset connection signal towards a matched packet (Inline Snort Final [34]). Thus, an inline Snort is categorized as Intrusion Prevention System (IPS) for its capability to drop packets, compared to an IDS where it is only capable to detect and alert without being able to drop the packets (Inline Snort [35]).

Ricciulli and Covel [36] integrated PF_RING with Inline Snort to accelerate the packet filtering process. PF_RING is a ring buffer inside the kernel to eliminate the need of copying packets from kernel to application space memory. Therefore, the Inline Snort can directly access this PF_RING buffer to process the packets without copying those packets to its memory space. The further discussion about PF_RING will be done in the later chapter.

Moreover, other applications can also benefit from the packet capture optimization. Another IDS named Suricata has also been ported to PF_RING. A network monitoring tool called Ntop natively utilizes PF_RING to accelerate its functions. In the future, more and more applications that require network packets can be modified to collaborate with PF_RING.

2.7. Current Packet Capturing Technologies

Braun et al. [37] made a comprehensive comparison study of different packet capturing solutions involving distinct hardware platform and operating systems (Linux and FreeBSD). They also made a modification on traditional Linux PF_PACKET packet capturing kernel module. They focused more on the improvements of software stack of packet capturing hierarchy to reduce packet loss.

Braun et al. [37] utilized commodity hardware since the recent hardware capability can capture packets at near wire speed within 1GE network. The significant hardware advancements consist of high speed bus system, multi-core processor, and network card with multi receiver (RX) queues. However, the hardware performance depends greatly on the software stack configuration. The software stack includes the network card driver, the packet capture module at the kernel, and the monitoring application at the user space. An example is given by comparison experiment done by Schneider et al. [38] who found AMD Opteron performed better than Intel Xeon in capturing packets, due to its distinct memory management and bus contention handling mechanism.

In initial packet capturing technology, the packets captured at the network card must be copied to the kernel buffer before they are further processed. The user space application that wants to use or analyze these packets must copy them to its memory space first. This mechanism in Linux is known as PF_PACKET system, while in FreeBSD, it is called BPF (Berkeley Packet Filtering).

However in Linux, the packets transfer from the kernel buffer to the application memory is done packet by packet, while in FreeBSD, it is accomplished by exchanging buffers contents; user space application reads packets from HOLD buffer, while the kernel uses STORE buffer to keep incoming packets. When HOLD buffer is empty, the STORE buffer will transfer its contents to it. Schneider et al. [38] recommend very big buffer size for these two buffers to reach decent packet capturing rate. An improvement to this is named ZCBPF (Zero Copy BPF); it implements memory mapping between both buffers, hence copying is not needed. However, Braun et al. [37] did not find significant performance differences between FreeBSD BPF and ZCBPF.

On the Linux side's improvement, Deri [39] and Dashtbozorgi and Azgomi [40] proposed a complement to PF_PACKET called PF_RING, which will be discussed in more details in the later section since this is the choice of our packet capturing platform.

The traditional packet capturing is that for every incoming packet in the network card, it will ignite hardware interrupt that will ask the kernel to take and process that packet. When the system gets overloaded with incoming packets, the kernel will be busy with issuing interrupts (interrupt storm), instead of copying the packets to kernel buffer. Related to interrupt storm, Schneider et al. [38] and Deri [39] have observed that capturing a 1GE stream with big sized packet (1500 bytes) is a lot easier than capturing it with small sized packet (64 bytes). This makes sense because smaller sized packet will generate more number of packets, therefore it will trigger more interrupts.

A solution to interrupt storm is to deploy device polling as researched by Mogul et al. [41], Kim et al. [42], and Rizzo [43]. Basically it is a scheduling approach to set a specific timer to when the interrupt should be generated to collect accumulated packets from the network card. In Rizzo [43] case, it also involves a scheduling to manage the sharing of CPU time between kernel and user space application. The optimization of device polling is also a part of our research topic, as the duration of polling is a part of our research parameters.

Specifically, Schneider et al. [38] compared the performance of device polling, between Linux and FreeBSD on AMD and Intel processor environment. He found out that Linux performed better packet capturing than FreeBSD by efficiently reducing the kernel's CPU time. Instead on FreeBSD, device polling decreased the packet capturing performance, even more it made the system unstable. To tackle this drawback, they recommend activating the network card's interrupt moderation. Furthermore, when they switched from device polling to traditional BPF, FreeBSD performed better, compared to Linux with traditional PF_PACKET plus device polling activated.

Another comparative experiment was accomplished by Deri [39] to validate his packet capturing method called PF_RING which was mentioned earlier. The experiment compared PF_RING with traditional Linux's PF_PACKET plus libpcap-mmap activated. The result showed that PF_RING method managed to improve the capture of small packets in the size of 64 bytes, and moderate packets in the size of 512 bytes. Additionally, this result was regenerated by Cascallana and Lizarrondo [44].

A later research by Deri and Fusco [45] combined PF_RING with TNAPI (Threaded-NAPI). TNAPI provides dedicated kernel thread for every receiving queue of the network card, thus it will not interfere with kernel threads for user space application. This experiment was proven to be able to capture a stream of 1GE small packets at wire-speed (1.488 million packets).

Braun et al. [37] evaluated the performance of Linux packet capturing in multicore processor environment, which is involving a PC with two Intel Xeon 2.8GHz clock speed, and another one with AMD Athlon 64 X2 5200+ 2.7GHz clock speed.They generated the input packets in a stream of 1GE traffic with 1.27 million packets per second (pps) that could be reached. This input stream was run for 100 seconds as much as five times. They did the experiments under Linux Ubuntu 9.04 with kernel version 2.6.32 and FreeBSD 8.0 operating system alternately.

Braun et al. [37] tells the drawback of running packet capture monitoring application's processes on two cores, which is the synchronization of memory between the two threads. This synchronization adds additional time load.

A part of Braun et al. [37] research was to compare the performances of the packet capture monitoring application under three different treatments; first treatment was if the kernel thread and the monitoring application were processed on the same core, the second was to run them on different cores, and the last was to let them automatically scheduled by the operating system's scheduler.

The result was that the most packets were captured when the kernel thread and the user space application were processed on the same core, with the condition that the input packet rates were low (around 550Kpps and below). While for the higher input packet rates, the performance was better when the kernel thread and the user space application's process were run on separate cores. And the worst performance occurred when the operating system's scheduler took place. Therefore, Braun et al. [37] concludes that both Linux and FreeBSD's scheduler are not accurate.

Another objective by Braun et al. [37] research was to compare packet capturing methods under low application load. They compared the traditional Linux PF_PACKET against PF_RING in transparent mode 2 (dedicated PF_RING capture), with the penetrating traffic at 1GE speed (±1 million packets/sec) over 100 seconds per run. They found out that increasing the size of the ring buffer (capture length), after a specific boundary did not improve packet capturing significantly. However, this finding is opposite to what is found by Schneider et al. [38]. Braun et al. [37] reaffirm that if the user space application cannot process the packets fast enough, then big buffer will even reduce the packet capturing performance. On the other hand, if the user space application is fast enough to process the packets, then big buffer will help to handle burst packets traffic.

The experiment results of Braun et al. [37] conclude that PF_RING outperformed PF_PACKET on both Intel Xeon and AMD Athlon 64 X2 5200+. Further analysis showed that PF_RING on this AMD platform performed better than when it ran on Intel Xeon.

Moreover, Braun et al. [37] also discovered that FreeBSD BPF performed better than Linux PF_PACKET, on both mentioned Intel and AMD platform. This finding confirms the claims of Schneider et al. ([38] and [46]).

In their next experiment, Braun et al. [37] compared the multi packet capturing applications among FreeBSD BPF, Linux PF_PACKET, and Linux with PF_RING. The Linux with PF_RING came as a winner as it had the best packet capturing performance, while FreeBSD BPF came at second and Linux PF_PACKET at the third. This discovery strengthens the results of Deri [39] and Cascallana and Lizarrondo [44].

Braun et al. [37] further added compression to the captured packets in their next experiment which consisted of two parts, by using packzip software. The compression level ranged from 0 (no compression) to 9, the higher compression level brings heavier application load.

The first part of experiment with compression compared FreeBSD and PF_PACKET on the mentioned AMD platform at both 256 and 512 bytes packet size stream. The results showed that at 256 and 512 bytes packet size stream, PF_PACKET performed better, with all capturing rates decreasing according to the increasing level of packets compression, which led to insufficient CPU time.

The second part of Braun et al. [37]'s experiment with compression involved comparison between FreeBSD, PF_PACKET and PF_RING at 64 bytes packet size stream, along with the previous compression when packets received. The results told that FreeBSD performed constant at all compression levels, but its performances were below PF_PACKET and PF_RING. PF_RING itself performed better than PF_PACKET at compression level 0 until 5, while at compression level 6 to 9, PF_PACKET came at first. However, packets compression is not a common practice in packet capturing applications, but this experiment result is to inform us what will happen if compression takes place.

At last, Braun et al. [37] give a significant contribution by proposing a signaling modification to PF_PACKET. They improve the device polling signaling mechanism of PF_PACKET. Therefore, instead of estimating a proper time-out, packets in the network card will only be collected when one of the below conditions occurs:

1. Specified number of packets are accumulated

2. Specified timer (microseconds) has been reached

The modified signaling mechanism of PF_PACKET by Braun et al. [37] managed to reduce system calls (kernel interrupt generations) shown by an experiment that used a stream of 64 bytes sized packets, which directly increased twice the rate of packets captured with no compression.

At their very last experiment, Braun et al. [37] combined TNAPI technology into PF_PACKET, their modified PF_PACKET, and PF_RING, and then compared them against each other. PF_RING combined with TNAPI produced the best packet capturing performance, followed by modified PF_PACKET combined with TNAPI, and the last came TNAPI combined plain PF_PACKET.

Finally, from all of the above comparative researches among packet capturing methods, we conclude that the configuration of packet capturing module has significant impact on the number of packets captured. This configuration setting should be adjusted according to the rate of incoming traffic, different load of traffic requires different treatment or setting.

We pick up PF_RING as a decent choice for our research platform, for its great performance as shown by the previous mentioned comparative studies, and its open source code availability. Thus, it is possible for us to integrate it with our methodology, which we will discuss in the later chapter.

2.8. High Speed Packet Capturing and Packet Filtering

Packet filtering can be done in several layers. Very commonly it occurs in Application layer or user space. The popular one for IDS application on Linux systems is Snort that resides at the Application layer. However, the way Application layer's IDS software works is by copying all network traffic from kernel to its memory then starts the filtering process as told by Deri [47].

Procedurally, the process of packet capturing by IDS at the Application layer is explained by Dashtbozorgi and Azgomi [40] as mentioned below:

1) When the packet arrives, the network card's firmware copies it into the memory called DMA (Direct Memory Access) then invokes the ISR (Interrupt Service Routine) that is implemented in the network card's driver, and then

this ISR removes the packet from the DMA memory and inserts it into the processing queue of a *softirq*.

2) The *softirq* (kernel) takes the packet from the processing queue and processes it.

3) The respective user application program will get the copy of the packet and then processes it.

The above system has some time duration issues related to the accomplishment speed of packet filtering. Those durations are:

1) Time required to invoke the interrupt

2) Time required to switch between the kernel space and IDS application

3) Time required to allocate memory

4) Time required to copy packet from the kernel space to the application space

5) Time required to process the packet's header

To reduce the above durations, Deri [39] and Dashtbozorgi and Azgomi [40] proposed PF_RING which is a network socket that bypasses the overhead processes in kernel level and captures the packets directly from the network card firmware and then copies the packets into a ring buffer inside the kernel. The IDS application then accesses this ring buffer via mmap (memory mapped) file I/O system call. However, there is still copying of packets from the network card to the ring buffer. This issue is then tackled by the implementation of nCap and DMA ring as explained by Dashtbozorgi and Azgomi [40].

nCap and DMA ring can connect directly the IDS application from the user space to the network card firmware. nCap is a network capture library based on kernel version 2.6 and Intel 1Gbps network card, while DMA ring is a wait-free queue structure of a DMA controller processor (DMA Ring [48]) which is based on 2.4 kernel version and 3COM 100Mbps network card. Therefore both nCap and DMA ring are not flexible methods since they are designed for specific kernel and network cards.

The research by Deri [47] explained that the packet filtering process can be accelerated by executing the filtering phase straightly at the kernel level. BPF (Berkeley Packet Filter) was recommended by his research to accelerate packet filtering since it inspects network packets that go through within the kernel. Furthermore, additional filtering can be added by installing another filtering application at the Application layer or user space level such as Snort, so that Snort will only handle potentially interesting packets because initial filtering will be done by BPF as explained by Deri [47].

The recent high speed network intrusion detection technique informed by Okamoto [49] is called "Virus Throttle" that stops fast worms with no affect on the normal network traffic.

2.9. Genetic Algorithm (GA) in Packet Filtering System

Genetic algorithm is a search method to find near optimum solution to a permutational problem, based on evolutionary natural selection process and genetics (Sastry et al. [50]). As in the definition, it implements on population or group of chromosomes of potential solutions which apply Charles Darwin's theory of survival of the fittest. This algorithm shall generate the best combination of solution for specific problem. For every set of generation, it will create a new set of estimations by selecting chromosomes depending on their level of fitness, and then these chromosomes will be bred using genetic operators that are inspired by natural genetics. This evolution is consequently expected to lead to better population.

The genetic operators are mentioned below according to description by Sastry et al. [50] and Shrivastava and Hardikar [51]:

1. **Reproduction**: It is an operator to make copies of better chromosomes in a new population, and it used to be the first operator implemented on a population. It chooses good chromosomes in a population to form a mating pool. It plays beneficial role to inherit the best of chromosomes of the previous population, thus the quality of the new population can be maintained or improved.

2. **Crossover**: An operator to recombine two chromosomes to get a better breed. The chromosomes are recombined by exchanging information which then new chromosomes are created based on this information, hence a hybrid could appear. However, the newly created hybrid may have either reduced or improved quality, thus to preserve some good chromosomes, not all of them are used in this crossover process.

3. **Mutation**: In this process, new information is added randomly to the search process. It maintains the diversity of population because of intensive operation of reproduction and crossover operator. It occurs at the bit level when the bits are being copied from the current to the new individual.

4. **Selection Method**: A method to assign more copies of solutions with better fitness values, in order to give them higher probability to survive. Generally, it consists of two categories, which are Fitness Proportionate Selection and Ordinal Selection. In our genetic algorithm process, we utilize Tournament Selection method that is a part of Ordinal Selection.

Note: Standard genetic algorithm pseudo-code can be found in Shrivastava and Hardikar [51].

The implementations of genetic algorithm have been proven to have a bright future in several engineering design cases as stated by El-Alfy [52]. He gives some examples where genetic algorithm was successfully implemented to tackle combinatorial optimization cases, in the Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP), Job Shop Scheduling (JSS), and Vehicle Routing Problem (VRP). Specifically on the paper by Shrivastava and Hardikar [51] the utilization of genetic algorithm in IDS scheme to decrease the percentage of false positive and false negative, and additionally to maximize the automatic response to attack, were described comprehensively.

Shrivastava and Hardikar [51] also inform that genetic algorithm can be used to analyze application patterns to assist IDS techniques, such as anomaly detection and misuse detection method. Shrivastava and Hardikar [51] report that the accuracy of genetic algorithm within IDS filtering is in average above 70% for data size ranging from 50 to 250 byte (50, 100, 150, 200, and 250). The accuracy results are in random pattern, which means the bigger the data size does not necessarily have lower or bigger accuracy.

Shrivastava and Hardikar [51] also prove that genetic algorithm consumes less computer memory compared to another search technique which is backpropagation neural network (BPN). This efficiency comparison results apply in all data sizes ranging from 50 to 250 byte (50, 100, 150, 200, and 250) with the average memory utilization in 32MB. With the maximum usable RAM (Random Access Memory) for a 32 bit system is 4GB. Thus, genetic algorithm process is considered safe for the computer memory resources.

Another genetic algorithm implementation within the network security is by El-Alfy [52] who used genetic algorithm to optimize the firewall rules accuracy. The goal of his research was to decrease the average number of comparisons against the firewall rule set, while at the same time preserving the precedence relationship among the rules. Firewall rules can create problem if a packet matches more than one rule. Thus, it is necessary to reorder the rules because a packet will be taken care based on the first action match. One rule precedes another. For example if we want to drop all UDP packets except from 192.168.113/24 subnet, then the rule that permits this subnet must be inserted before the wildcard rule that blocks all UDP traffics.

El-Alfy [52] also implicitly states that genetic algorithm is fast to deploy because it is flexible for global optimization, and does not require the programmer to have the understanding of problem specific knowledge to find an acceptable solution. El-Alfy [52] compared genetic algorithm against other two firewall optimization method namely branch-and-bound (BB) method that is an exact solution approach, and the sorting method that uses an approximate approach. These all three methods were tested under four probability distribution techniques which were *Eq, Unif, Zipf, and r-Zipf.* Ten rules with several precedence relationships were provided to go through the tests of maintaining the precedences and examining the average number of comparisons.

The above comparisons results by El-Alfy [52] concluded that genetic algorithm and BB method performed the best improvement (less average number of rule comparisons) under uniform distribution (*Unif* and *r-Zipf*) environment compared to the sorting method, with the *r-Zipf* environment gave the higher percentage of improvement than *Unif.*

Special for genetic algorithm technique, besides the ten rules used, within the reproduction process it also used crossover operator with single point and 0.8 probability out of 1. Another operator used was the swap mutation operator with 0.1 mutation probability out of 1. These parameters values are commonly used in genetic algorithm experiments with good outcomes as told by El-Alfy [52].

On the other hand, the two other comparison experiments each with complete/full precedence relationships and no precedence relationships showed that the three optimization techniques (BB, genetic algorithm, and sorting) performed in the same level with the same average number of rule comparisons. For the case with complete/full precedence relationships, the average number of rule comparisons is 7.78 for all three optimization techniques and the original non-optimized one. In other words, optimization does not make sense. However, for the case with no precedence at all, the average number of rule comparisons is 3.2160 for all three optimization techniques and for the original non-optimized one is 7.7840. Therefore, optimization is worth doing. Overall, the research by El-Alfy [52] concluded that genetic algorithm was proven to be a decent choice to optimize the rewriting of firewall rules in the environments where the precedence relationships exist either partially or completely.

One more distinct genetic algorithm related research was by Nottingham and Irwin [53]. They made an effort to accelerate packet classification in parallel manner by taking advantage of the parallel computation capability of the current commodity GPU (Graphics Processing Unit). The meaning of packet classification itself is to classify a packet based on the header information and then pin-point it to the correct receiving application in real time. The exploitation of the GPU parallel computation itself is synchronized with the parallel behavior of the genetic algorithm, specifically they used General Processing (GP-GPU) genetic algorithm.

The research objective of Nottingham and Irwin [53] is to "produce contextually optimized filter permutation in order to reduce redundancy and improve the per-packet throughput rate of the resultant filter program." The need for genetic algorithm in filter permutation optimization rises from the limitation of permutation optimization using adaptive pattern matching algorithm. This limitation occurs when the number of filters grows large (above 20) as stated by Tongaonkar [54].

While the mechanism of packet classification by Nottingham and Irwin [53] itself is to compare a subset of packet header information to a set of static values, to determine the type of the respective packet, hence it can be forwarded to the destination application. Anyway, just like in firewall rule matching redundancy problem, packet classification filter rules might also match a packet with more than one rule, thus the order of filters is significant.

The effectiveness of permutation optimization is limited if the filter rule grows in size. Nottingham and Irwin [53] bred a near-optimal permutation of filter rules utilizing a generation confined, GP-GPU based genetic algorithm. They utilized OpenCL as the GPU programming language for its parallelism and wide portability with GPU card architectures and processor types. The respective genetic algorithm was purposely created to improve permutations of filter, so that it can minimize the control flow graph for the network context, hence the packet classification process duration can be as brief as possible.

As mentioned above that Nottingham and Irwin [53] utilized the parallelism of the GPU, thus they executed the experiment over multiple GPU cores using a Single Program Multiple Data (SPMD) setting. A chosen parameter was the number of individual nodes of the filter tree in the resultant control flow graph, because it reflects the shortest path. The packet classification itself ran by recording the sliding window of recent network flow. They determined which chromosome of the permutations that caused the shortest control flow graph structures for specific network context. After the best chromosomes or factors had been found, then they bred these in parallel permutations, and finally select the permutation with the best performance to use as the classifier. This process was executed multiple times at given intervals to adjust to the ever changing network characteristics without user interaction required. The run duration for this breeding phase depends on the chromosome population size and the number of generations created (the frequency of evolution).

The first set of tests by Nottingham and Irwin [53] involved ten randomly generated filter programs with every filter program consisting of 20 different filters, and this test's population size was 100 chromosomes that were run for 200 generations. The results were awesome with 2.5 times up to 10 times of node count reduction from unoptimized packet classification, and fast processing time ranging from 36 to 78 seconds.

The second test set involved six randomly generated filter programs with each filter program containing 50 different filters, the chromosome population was 250 which were run for 500 generations. The results generated showed significant reduction of node count ranging from 9 to 12 times from the unoptimized packet classification. However, the processing durations were quite high ranging from 1744 to 2115 seconds, but this can be further decreased by taking advantage of the GPU parallel computation (Nottingham and Irwin [53]).

Another research by Nottingham and Irwin [55] was trying to optimize packet filtering that classifies packets based on arbitrary packet header information, thus the packets can match their desired destinations. They used genetic algorithm approach to optimize filter permutation and to eliminate redundant codes, so that the average path length of the filter control flow graph can be reduced. The permutation optimization matters since the overlapping of rules comparison is highly possible. Thus Nottingham and Irwin [55] conclude that the discovery of optimum filter permutation is equivalent to the discovery of directed acyclic control flow graph.

The simulation steps taken by Nottingham and Irwin [55] are listed below:

1. Construct simple abstract control flow graph generator that accepts arbitrary permutation of a set of filters. The predicates or attributes of these filters are set up with occurrence percentage value to make sure some packets will match these filters.
2. Convert the permutation of filters to a control flow graph.
3. Reduce the control flow graph length by applying predicate assertion propagation and static predicate prediction. Predicate assertion propagation works by evaluating an edge of predecessor's dominator node. If it points to the subsequent's node predicate whose result can be determined by the predecessor's dominator set, then it means the subsequent filter's predicate node is a redundant, because the edge of predecessor filter's node can directly reach the child node of the subsequent filter. While static predicate prediction complements this by taking

the implicit converse of predicate assertion propagation's result, therefore any opposite or false predicate match will be removed from the path.

The fitness value of each chromosome (filter permutation) itself is determined based-on the number of eliminated nodes (higher better).

However, the best performers are not automatically selected since the selection method used is Roulette Wheel selection. It divides a wheel into areas of chromosomes. The size that it allocates to each chromosome (filter permutation) reflects their fitness values. It will randomly pick up a value between zero and the grand total fitness value from all chromosomes to decide which chromosome to select. Roulette Wheel method is chosen to both achieve near optimal solution and to explore the solution space.

4. Do crossover to create one or two child chromosomes. However, it might be profitable to reserve some parent chromosomes from intact, therefore the percentage of crossover occurrence can be specified. In this case, Nottingham and Irwin [55] applied 70% of crossover occurrence.

   The formula of their crossover is described below:

   a. Copy the first part of $\frac{n}{2}$ of the first parent to the first child's first part of $\frac{n}{2}$ chromosome, then copy the remaining of the first parent's chromosome to the first child in the order they appear in the second parent.

   b. For the creation of the second child, the order of parents is reversed.

5. Apply the probability of mutation which is the alteration of chromosome genes. It is done by swapping one random gene index with another random gene index from a different chromosome.

6. Compare the node count (attributes/predicates) of two filter permutations (control flow graphs) from the same set of filters. One of the filter permutations is generated by genetic algorithm.

The above procedure was applied to three unique test beds; the first test bed involved ten filters, each filter consists of five to ten predicates with the condition that 80% chance a filter predicate will contain "2==K". This test bed was executed ten times with population size of 100 and 200 generations. The result is that the genetic algorithm managed to reduce the node count by 20%.

The second test bed involved fifty filters with ten to twenty predicates per filter, population size of 100 and number of generations of 200, additionally with 80% chance that a filter predicate will contain "2==K". This test bed was executed six times with the significant result of node count reduction by 40% in average. However, it generated increased computation time as the effect of more number of filters.

The last test bed included implicit node count redundancies with fifty filters, ten to fifteen predicates per filter, population size of 250, number of generations of 500, and pure random probability of predicates contents. This test bed was executed six times with the best result compared to the previous two. The percentage of node count reduction reaches 50% in average with highest computational time compared to previous sets.

From the overall experiment by Nottingham and Irwin [55], we can see that genetic algorithm is highly effective in tackling permutation case with the node count reduction ranging from 20% up to 50%. The effectiveness of genetic algorithm in this case is linear to the number and complexity of the filters, the number of population, and the number of generation. The trade-off comes from the increased computational time that results from the increasing complexity, number of filters, also the population size and number of generation.

However, the research simulation by Nottingham and Irwin [55] still has some rooms for potential improvements; the content of the predicate can be extended from the simple string e.g. "2==K" to become more realistic, even reflecting the real firewall rule content, for example, protocol type or source IP address. The experiment may also be brought into real environment instead of mere simulation program. Furthermore, their work can be extended by further accelerating the filter optimization process. This can be accomplished by fine tuning the packet classifier or the packet capture's software module, thus an accelerated packet capture's module shall automatically speed up the filter optimization process.

From the above discussed results, we can conclude that genetic algorithm has potent impact towards the packet classification or in general term, it will perform well in packet capturing area.

2.10. Network Penetration Testing

Network penetration testing is an act to generate flow of packets towards the packet receiver. The packet receiver itself is a computer system able to capture and analyze the packets. As discussed in the previous sections, it consists of hardware and software stack. The hardware includes the network card and the processor, while the software stack is the kernel level packet processing.

In this research, penetration testing is meant to measure the fitness values of our genetic algorithm optimized packet filtering framework. The penetration testing that we use in our research is ping test. It is sufficient for us, because we focus on packet capturing, instead of packet filtering. We deploy different levels of ping load, which is based on its packet size. Our packet capturing framework will try to reach the near optimum packet capture setting, towards the different loads of penetrating traffic. The measurement is so far based on the calculated packet throughput. The throughput is calculated after every seven seconds of packet capturing process. Seven seconds of duration is chosen because the common network card speed nowadays is 100Mbps. Hence for seven seconds, the rough packet size captured is 700Mb, which equals to 87.5 MB. In real packet transmission, it reflects a small to medium sized file, which is suitable for a kick-start point in this research.

Another purpose of the penetration testing is to examine the security of the network. Security testing needs to be conducted to check whether the security policies are working properly and whether new attack techniques can be detected or not. Chen and Laih [56] define penetration testing as a live test over an enterprise network.

Penetration testing is conducted by a Security Auditor (SA) while real intrusions are launched by hackers. The techniques and tools are most probably the same, but the purposes are different. Chen and Laih [56] were concerned about the identification to separate penetration packets coming from the auditors and from the malicious hackers. To solve this identification problem, they proposed a new IDS function called IDSIC that stands for Intrusion Detection System with Identification Capability.

Chen and Laih [56] add fingerprint information to the penetration packets. Two fingerprint components would work to accomplish the fingerprint evaluation, those two components were fingerprint adder and fingerprint checker. The fingerprint adder will hash the packet message, encrypt the hash value, and then concatenate this hash to the message. After that, the fingerprint adder will add this modified message into the packet meant for penetration testing.

When IDSIC-based receiver receives the penetration packet with fingerprint, its fingerprint checker will validate the fingerprint, and then it will verify the hash value. If the hash value is correct, then this packet is categorized as penetration testing by Security Auditor, therefore alert will not be triggered. Or if the hash does not match, then it will generate an intrusion alert.

In order to do penetration testing, packet can be crafted to model hidden attacks (metamorphic and polymorphic shellcodes) using engines such as those identified by Okamoto [49], which are AMDmutate, CLET, alpha mixed, alpha upper, call4 dword xor, contextcpuid, count-down, fnstenv mov, jmp call additive, and shikata ga nai. Metamorphic shellcode is a malicious shellcode that is disguised by an encoder while polymorphic shellcode is a malicious shellcode that is hidden by an encryption (Okamoto [49]).

2.11. Summary

In this chapter, we have given a fundamental of computer networks principles and its security problems and mitigations (firewall rules and IDS). We have also reasoned our choice for PF_RING as packet filtering module for this research. Braun et al. [37], Schneider et al. [38], Deri [39], Dashtbozorgi anf Azgomi [40], Cascallana and Lizarrondo [44], and Deri and Fusco [45] have found the reliability of PF_RING in their researches. The uses of genetic algorithm in computer networks area have been found useful in packet filtering research by Shrivastava and Hardikar [51] and Nottingham and Irwin [55], in firewall rules accuracy research by El-Alfy [52], and in packet classification research by Nottingham and Irwin [53]. To measure our proposed packet filtering framework, we utilize ping traffic test, which is a part of penetration testing techniques. Ping test can go sequentially from small packet size (64 bytes) to the big packet size, for example in Ethernet standard, the Maximum Transmission Unit (MTU) is 1500 bytes. This is useful to discover the specific treatments for every traffic load. The other penetration technique is crafted packet penetration test, which is beneficial to evaluate the security policy of the network gateway e. g. a firewall. The security policy includes firewall rules and IDS content filtering.

CHAPTER 3

METHODOLOGY

3.1. Introduction to Genetic Algorithm Library

3.1.1 GAlib

Sastry et al. [50] recommend using existing genetic algorithm for initial implementation, because it is fast and there are also ready-to-use genetic algorithms library, which can be found on the Internet. GAlib is one recommended by them.

GAlib written by Matthew Wall of Massachusetts Institute of Technology (MIT) is a free set of C++ genetic algorithm objects (GAlib [57]). It contains functions utilizing genetic algorithms to optimize any C++ application. It is capable to provide most problem representations and genetic operators. GAlib can be compiled on various UNIX-based operating systems and Windows.

GAlib's capabilities include:

1. Generation of random number as a base for gene's randomized value. The one being used is the generation of random number within the range 0-1.
2. Ability to evolve populations and/or chromosomes in parallel by utilizing multiple processors.
3. The parameters to become the genes can be set in a text file, from a command line argument, or in another code file.
4. Support overlapping and non-overlapping populations.
5. Customizable genetic algorithm techniques and attributes, such as initialization method, mutation method and probability, crossover method, comparison method, termination method, speciation method, elitism, replacement strategy, selection method, and statistics recording.

3.1.2 GAlite the Minimized GAlib as The Methodology Platform

Our research focuses on the methodology to optimize packet capturing. We do not emphasize on the genetic algorithm itself. A simple existing genetic algorithm library would be enough to kick start our experiment, to evaluate the effectiveness of our methodology or framework. Therefore, we utilized a minimized version of GAlib called GAlite (GAlite [58]).

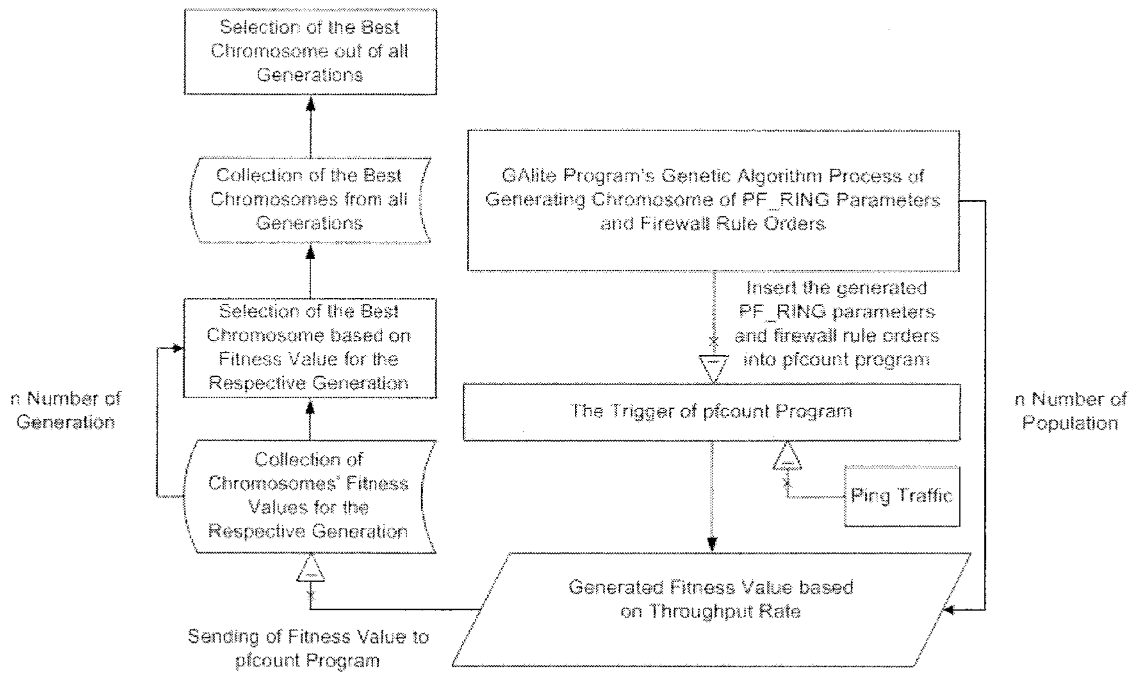
We started the GAlite implementation with 50 population size, 50 generations, 0.2 probability of mutation, 0.4 probability of crossover, and tournament selection method. In this tournament selection, two chromosomes are picked up randomly, then their fitness values are compared, and after that the winner is chosen to be the next individual. This process repeats until the specified number of population is achieved (Sastry et al. [50]).

We modified a sample GAlite application which is adjusted to our permutation case, i.e. to optimize the configuration and firewall rules ordering of our modified PF_RING application. This GAlite sample application is simply named ga.

Our permutation case is based on PF_RING parameters, which are discussed in section 3.3 later. These parameters are permutated inside our GAlite application. Each permutation of PF_RING parameters is a chromosome/individual. And then, the parameter values generated will become the inputs of our PF_RING application. Our PF_RING application will run for 7 seconds with every chromosome, and then generates a fitness value based on the total number of packets captured (throughput rate). This process repeats according to GAlite's algorithm and attributes (number of generations, population size, etc).

Finally, the best chromosome is obtained from each generation, which are then compared to each other to select the best chromosome. The selected best chromosome is the near optimum combination of PF_RING configuration and its firewall rules ordering.

Our overall packet capturing optimization and firewall rules reordering methodology framework is depicted in the below flowchart:



Selection of the Best Chromosome out of all Generations
Collection of the Best Chromosomes from all Generations
Selection of the Best Chromosome based on Fitness Value for the Respective Generation
Collection of Chromosomes' Fitness Values for the Respective Generation
n Number of Generation
GAlite Program's Genetic Algorithm Process of Generating Chromosome of PF_RING Parameters and Firewall Rule Orders
Insert the generated PF_RING parameters and firewall rule orders into pfcount program
The Trigger of pfcount Program
Ping Traffic
n Number of Population
Generated Fitness Value based on Throughput Rate
Sending of Fitness Value to pfcount Program

Figure 3.1. Genetic Algorithm Optimized Packet Filtering Framework

3.2 Packet Capture Platform to be Optimized by Genetic Algorithm

A PF_RING definition by TNAPI [59] states that "PF_RING is a Linux kernel patch that allows packet capture to be improved". This packet capture improvement is accomplished by bypassing some kernel network layers and replacing system calls such as read() with DMA.

PF_RING is the selected packet filtering acceleration module for this research because it does not depend on specific network card type (hardware independent). Thus, it is easier to integrate into the operating system compared to nCap and DMA ring. The reason of its independence over hardware is because it resides in the kernel at the bottom of networking stack above NAPI (New API). NAPI is a linux kernel feature to reduce the number of interrupts requested and to decrease the latency. It disables interrupts in high traffic and the packets are collected by polling to decrease the system load created from interrupts processing, and while the traffic load is low then the interrupt scheme activates again to minimize the latency from the pooling waiting process (Leitao [60]). NAPI was firstly built for linux kernel version 2.6 and was subsequently made compatible with kernel version 2.4.20.

PF_RING supports libpcap to integrate existing pcap-based applications. It can also filter the packets by inspecting the header and or the payload contents in addition to BPF. Plugins may be added for advanced packet parsing and content filtering (NTOP PF_RING [61]). Moreover, PF_RING is able to work in transparent mode as explained by PF_RING Transparent Mode [62]. Three modes can be chosen if transparent mode is activated. First is when transparent_mode = 0, which means packets are sent to PF_RING with standard kernel procedure where the packets are also sent to other kernel components. Second is when transparent_mode = 1, where the packets are delivered to PF_RING by the NIC driver. The packets are also sent to other kernel components but it would be faster since it skips traditional kernel processing. The last mode is when transparent_mode = 2, it is similar to the previous mode except that packets are not forwarded to other kernel components. Thus this mode performs best in term of speed. The packets are then discarded after being captured and analyzed.

Below are types of PF_RING module variations with their advantages and drawbacks.

a. **PF_RING DNA (Direct NIC Access)**: It is a scheme to map NIC memory and registers to application layer, where the copying of packets from the NIC to DMA ring is accomplished by NIC NPU (Network Process Unit) instead of NAPI-based CPU. Therefore the CPU cycles are reduced (PF_RING DNA [63]). Additionally, the DNA driver requires a license (paid) which is based on the driver speed of either 1Gbit or 10Gbit.

b. **PF_RING TNAPI (Threaded NAPI)**: TNAPI scheme complements the MSI-X technology of current Ethernet cards that partitions the incoming RX (receive) queue into several RX queues according to the number of CPU cores (one RX queue per core) (TNAPI [59]). The traffic management will balance the flows according to which core they initially belong to, thus each packet from the same flow will be forwarded to the same core instead of going in round-robin manner. This mechanism is made possible by utilizing RSS (Receive Side Scaling) and is only applicable to RSS-enabled network cards (RSS [64]).

Furthermore, TNAPI [59] concludes that for its efficient traffic flow distribution, TNAPI module is considered to be scalable because the more CPU cores the more RX queues it can accommodate and the speed of packet capture itself will improve since the packets are fetched simultaneously from every RX queue. Moreover, it collaborates with PF_RING to bridge the RX queues to application layer/user space, so that an application can spawn one thread for every queue.

3.3. PF_RING Parameters to be Optimized by Genetic Algorithm

We modify an existing PF_RING based application named pfcount to activate and monitor the packet capturing process. The permutations on PF_RING parameters which we have chosen, based on the official PF_RING manual by NTOP [65] are coded in pfcount application. Those parameters are listed below:

1. **caplen or snaplen** = the size of packet capture length. The range is between 64 bytes upto the size of Maximum Transmission Unit (MTU) of the layer 2 protocol. In this research, the protocol is Ethernet with its MTU of 1500 bytes. But in our algorithm, the maximum MTU is 1536 bytes to accommodate the increase of MTU for every 64 bytes, and also helps to simplify the rounding of result.

2. **active_wait** = ingress packet wait mode whether passive (poll) or active wait. When active wait takes place, every packet will activate kernel interrupt. While in passive wait, packets will be polled based on timer before kernel interrupt is invoked to process the packets. Integer 0 represents active wait and 1 represents passive mode as coded in PF_RING application.

3. **watermark** = packet poll watermark whether to make it low (reduce latency of poll but increase the number of poll calls) or make it high (increase the packet latency but reduce the number of poll calls). The range is 1 byte (check everytime for packet) until 50% of the maximum ring buffer size. In this research, the maximum ring buffer size is 4096 bytes, thus the maximum watermark value is 2048 bytes.

4. **cluster_type** = cluster type whether is per-flow or round-robin. In per-flow cluster type, each part of a packet transmission flow will go to the same cluster of the buffer. While in round-robin cluster type, the parts of a packet transmission flow might go to the different cluster of the buffer.

5. **poll_duration** = poll timeout in msec that takes care of data structures synchronization. According to Dovrolis et al. [66], an acceptable clock interrupt delay for real time applications is around 100 ms to 200 ms. Additionally, Linux Hardware Interrupt [67] explains that the Linux system timer is programmed to generate a hardware interrupt 100 times in every second, which means that each hardware interrupt will take place every 10 ms. This concept has been affirmed by Timing in the Linux Kernel [68] that system timer ticks every 10 ms. Therefore, when passive wait is chosen, the appropriate range of poll duration is 10 ms – 200 ms with escalation every 10 ms.

6. **Firewall rule set** = a set of firewall rules in this research consists of 13 rules. Each rule will be given a randomly generated priority value. This firewall rules ordering mechanism applies in condition where there are more rules to accept packets than the rules to drop them, since the only fitness value used is throughput rate. It is also a good method to reorder the rules, if the network administrator is faced with loads of existing firewall rules. The rules indexed in the chromosome start from 5 – 18.

3.4. PF_RING Application's Representation of Chromosome

Every chosen parameter is part of a chromosome which is indexed starting from zero (0). In other words, each parameter is a gene. In standard genetic algorithm, every gene contains a float range value from 0-1 that will be randomly generated. Thus, specific formula for each parameter must be created to accommodate their real values from minimum to maximum. Below are the formulas based on C programming syntax, where gene(x) is the float random number generator ranging from 0-1:

1. **snaplen** = (gene(0) x 23) x 64 + 64

2. **active_wait** = (gene(1) > 0.5) ? 1 : 0

3. **watermark** = (gene(2) x 89) x 23 + 1

4. **cluster_type** = (gene(3) x > 0.5) ? “cluster_round_robin” : “cluster_per_flow”

5. **poll_duration** = (gene(4) x 19) x 10 + 10

6. **firewall rule [ ] priority** = gene(5)

However, due to 7 significant digits of C++ float data type of gene(x), the generated value of the above mentioned parameters might exceed their specified limit. If this occured, then a specified maximum value has to be taken.

The above mentioned parameters are the input files for a PF_RING based application called pfcount to count the number of packets received by PF_RING kernel module. Importantly, the fitness value to evaluate every chromosome is the throughput or the total number of packets received. The higher the fitness value the better it is.

The pseudo code of pfcount represents the mechanism of PF_RING itself, especially where the above chosen parameters take place. The pseudo code is listed below.

```
Read snaplen input;
Read wait mode input {
        If (0)
                Active wait (Read poll duration = 0);
        If (1)
                Passive wait (Read poll duration from input; }
Read watermark input;
Read poll duration input {
        If (Active wait)
            Poll duration = 0;
        Else
            Poll duration = input;
Read cluster type input;
Read firewall rules order input;
PF_RING Socket Initialization;
While (true) {
         If (watermark = watermark input)
        Read incoming packets;
        Process incoming packets;
        Filter Incoming Packets;
        Count received packets;
        Generate throughput value;
    Else (poll duration times out)
        Read incoming packets;
        Process incoming packets;
        Filter Incoming Packets;
        Count received packets;
        Generate throughput value; }
```

3.5. Summary

This chapter discusses about our genetic algorithm optimized packet filtering methodology which covers GAlite (minimized GAlib) as our chosen genetic algorithm library and PF_RING as our packet filtering platform. The PF_RING kernel parameters and firewall rules are genes that make up a chromosome, with each gene's value (kernel parameters) is generated based on their assigned formula, while the firewall rules orders are evolved. The GAlite application will call/trigger PF_RING application 'pfcount'. The generated PF_RING kernel parameters and firewall rules ordering by GAlite application are the inputs for pfcount. pfcount will be penetrated by different loads of ping traffic, which after that, it produces throughput rate. This throughput rate is sent to GAlite application to be used as fitness value. At the end, the best chromosome from every generation is generated. Each of these chromosomes are the optimal PF_RING kernel parameters and firewall rules ordering.

CHAPTER 4

EXPERIMENTS RESULTS AND ANALYSIS

4.1. Experiment Software and Hardware Environment

The experiment was conducted based on the following hardware and software environment:

1. Virtual Box 4.1.6
2. CPU: AMD Athlon Neo X2 Dual Core Processor L335 1.60GHz
3. Number of Virtual CPU: 1
4. Acceleration: AMD-V Hardware Virtualization with nested paging enabled
5. Base Random Access Memory: 512MB
6. Network adapter: Realtek PCIe GBE Family Controller 1Gbps
7. Linux Kernel version: 2.6.32
8. OS: Debian 6.0.4
9. PF_RING 5.4.5

The above specification uses moderate hardwares so we can stress it out, in order to measure the optimal configurations. The chosen Linux platform and kernel version are the ones compatible and stable to be used with the then stable PF_RING 5.4.5.

The packet capturing framework was measured against twenty-four levels of ping traffic load according to the size of each packet injected (increment every 64 bytes), which are from 64 up to 1536 bytes. These are considered low, medium, to high network traffic load.

The initial genetic algorithm properties itself generates 50 of population size, 50 generations, 0.2 of mutation probability, 0.4 of crossover probability, and tournament selection method. In tournament selection, two chromosomes are picked up randomly, then their fitness values will be compared. After which the winner will be chosen to be the next individual. This process repeats until the specified number of population is achieved (Sastry et al. [50]).

The second experiment involves genetic algorithm properties that evolve around 100 population size, 100 generations, 0.4 of mutation probability, 0.5 of crossover probability, and tournament selection method as in the initial experiment.

The procedure related to the above mentioned genetic algorithm properties has been explained previously in section 3.1.2.

4.2. Experiment Results and Discussion

4.2.1. Results and Discussion of Initial Experiment

The best chromosomes were determined from every traffic load. Below are the graphs and table showing the throughput progression for each load, and the patterns of the selected PF_RING parameters.

Table 4.1 The Chosen Best Chromosome Contents for Different Traffic Load Levels

No.	**Input Packet Size (bytes)**	**Best Chromosome Throughput (bytes/7 sec)**	**Snaplen (bytes)**	**Active/ Passive Wait**	**Watermark Value (bytes)**	**Cluster Type**	**Poll Duration (ms)**	**Firewall Rules Order**
1.	64	4435.92	192	Active wait	1703	Per flow	10	5-1-8-9-13-3-7-10-12-2-4-11-6
2.	128	6457.92	384	Active wait	2920	Per flow	350	11-8-7-3-12-2-6-13-1-9-5-4-10
3.	192	8640	1664	Active wait	990	Round robin	210	12-9-6-2-3-11-5-4-8-1-10-13-7
4.	256	10353.2	192	Active wait	1864	Per flow	90	12-10-1-5-3-13-11-4-2-9-7-8-6
5.	320	11975	512	Passive wait	2644	Per flow	210	5-8-7-3-9-4-1-6-2-13-12-10-11
6.	384	13465	1728	Active	1864	Per flow	170	13-1-7-3-

				wait				8-12-2-6-5-9-4-10-11
7.	448	15139.5	384	Active wait	942	Per flow	100	3-1-12-10-7-13-2-6-9-11-8-4-5
8.	512	17259.4	1088	Active wait	114	Per flow	150	7-1-9-2-3-13-5-10-4-12-6-11-8
9.	576	19172.2	2304	Active wait	183	Per flow	30	4-6-1-10-13-9-2-8-5-3-12-11-7
10.	640	21226.7	2048	Active wait	714	Per flow	100	11-2-4-5-9-12-7-6-8-13-10-3-1
11.	704	22433.3	3840	Active wait	574	Round robin	90	10-6-13-9-5-4-7-11-12-8-1-3-2
12.	768	24713.9	640	Active wait	369	Per flow	110	12-10-11-13-5-9-7-4-6-2-1-8-3
13.	832	26786.2	2432	Active wait	1588	Round robin	230	6-12-11-1-4-10-8-5-2-7-13-3-9
14.	896	28549.3	2240	Active wait	921	Per flow	280	2-12-3-8-13-11-10-6-7-5-1-4-9
15.	960	31112	1088	Active wait	760	Per flow	110	3-1-10-6-9-2-11-12-13-4-8-5-7
16.	1024	33081	192	Passive wait	1816	Per flow	130	12-6-7-3-13-9-1-11-5-4-8-10-2
17.	1088	34655	1920	Active wait	1519	Per flow	150	2-9-1-13-11-5-12-3-4-7-10-6-8
18.	1152	36421	704	Passive wait	3426	Per flow	70	3-1-13-6-11-12-2-9-8-7-5-4-10
19.	1216	40131	2560	Active wait	829	Per flow	40	2-4-6-12-9-3-11-10-7-8-1-5-13
20.	1280	40461.1	2432	Passive wait	1103	Per flow	120	4-2-13-8-12-3-6-5-7-10-9-1-11

21.	1344	42172.2	576	Passive wait	3081	Per flow	10	9-10-13-11-4-2-3-5-6-8-1-7-12
22.	1408	46237.6	64	Passive wait	369	Round robin	90	6-8-10-9-1-4-11-12-5-7-13-3-2
23.	1472	47368.8	384	Active wait	1496	Per flow	200	9-8-6-7-11-10-5-2-12-3-1-4-13
24.	1536	50485.1	448	Active wait	1034	Round robin	80	11-12-10-6-3-9-8-2-7-4-5-1-13

The above Table 4.1 displays the best chromosome generated from every ping traffic penetration load. It includes the highest recorded throughput captured, the best selected PF_RING kernel parameters values, and the firewall rules ordering. The description of PF_RING parameters have been given in section 3.3. We can take for example, in the first row for input traffic 64 bytes with highest throughput 4435.92 bytes/7 sec, the best PF_RING configuration (chromosome) recorded states that the best snaplen (capture length) size is 192 bytes with active wait used (no device polling), the watermark value is 1703 bytes, per flow cluster type, and poll duration 10ms (unapplied because polling is only for passive wait mode). While the firewall rules ordering for this input traffic sequentially puts 5-1-8-9-13-3-7-10-12-2-4-11-6 in the order. For some other input traffics, passive wait (device polling) is used, hence poll duration is activated.

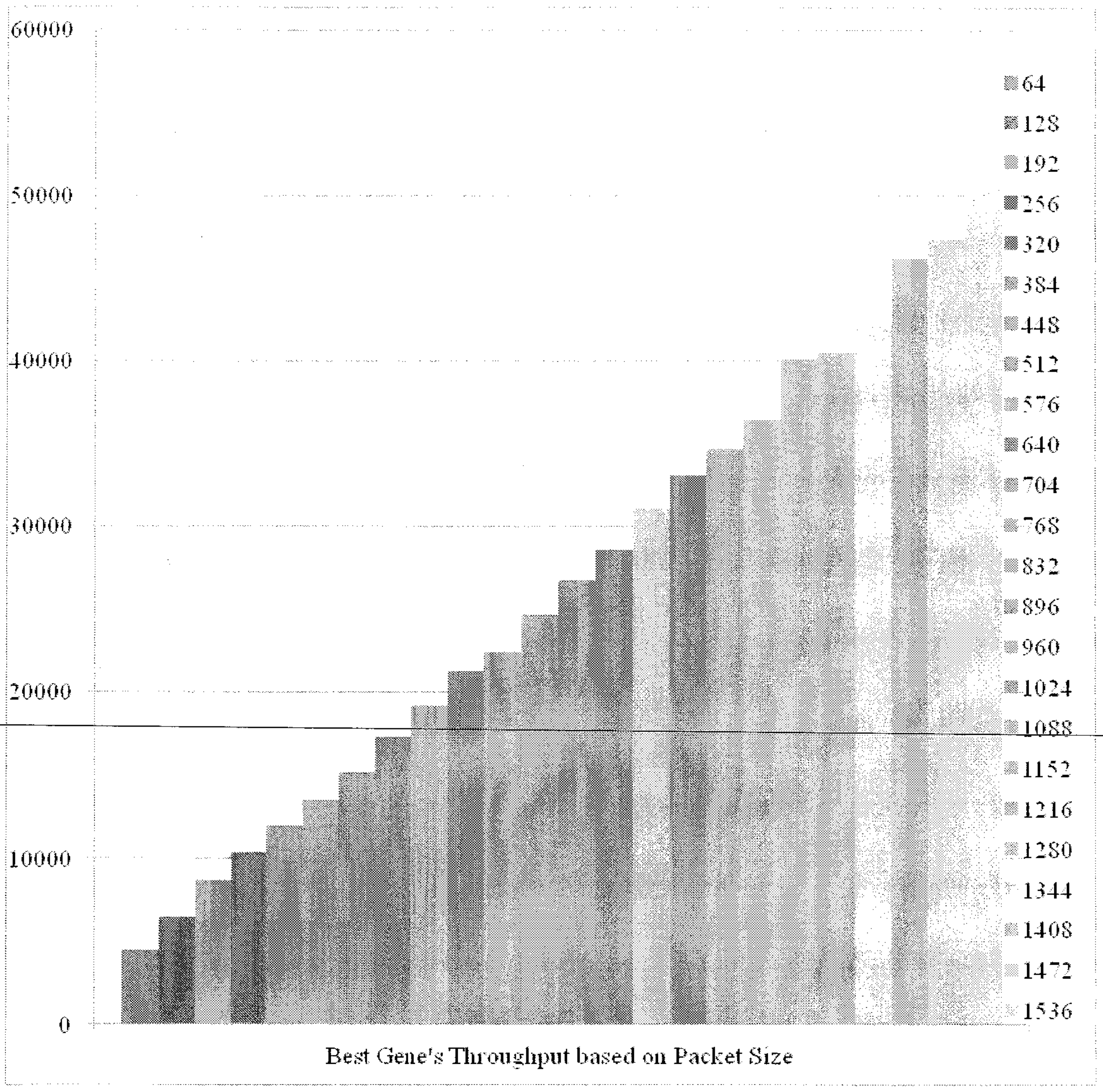


Figure 4.1 Throughput Progression from Different Traffic Loads

Figure 4.1 and Table 4.1 show the sequential throughput rate from ping traffic penetration, starting from 64 until 1536 bytes of packet size. The throughput rate progression is stable from 64 until 1152 bytes of packet size with moderately constant progression. But starting from 1216 until 1536 bytes of packet size, the throughput rate started to progress with more unstable varieties in progression. This shows that optimization is mostly needed within this range. The former stability affirms the research result by Deri [39] that claims PF_RING performs consistent at low packet sizes. In Figure 4.2, 4.3, and 4.4 below; the snaplen size, watermark value, and poll duration for each packet size have high significant difference. However, they do not block the consistency of PF_RING.

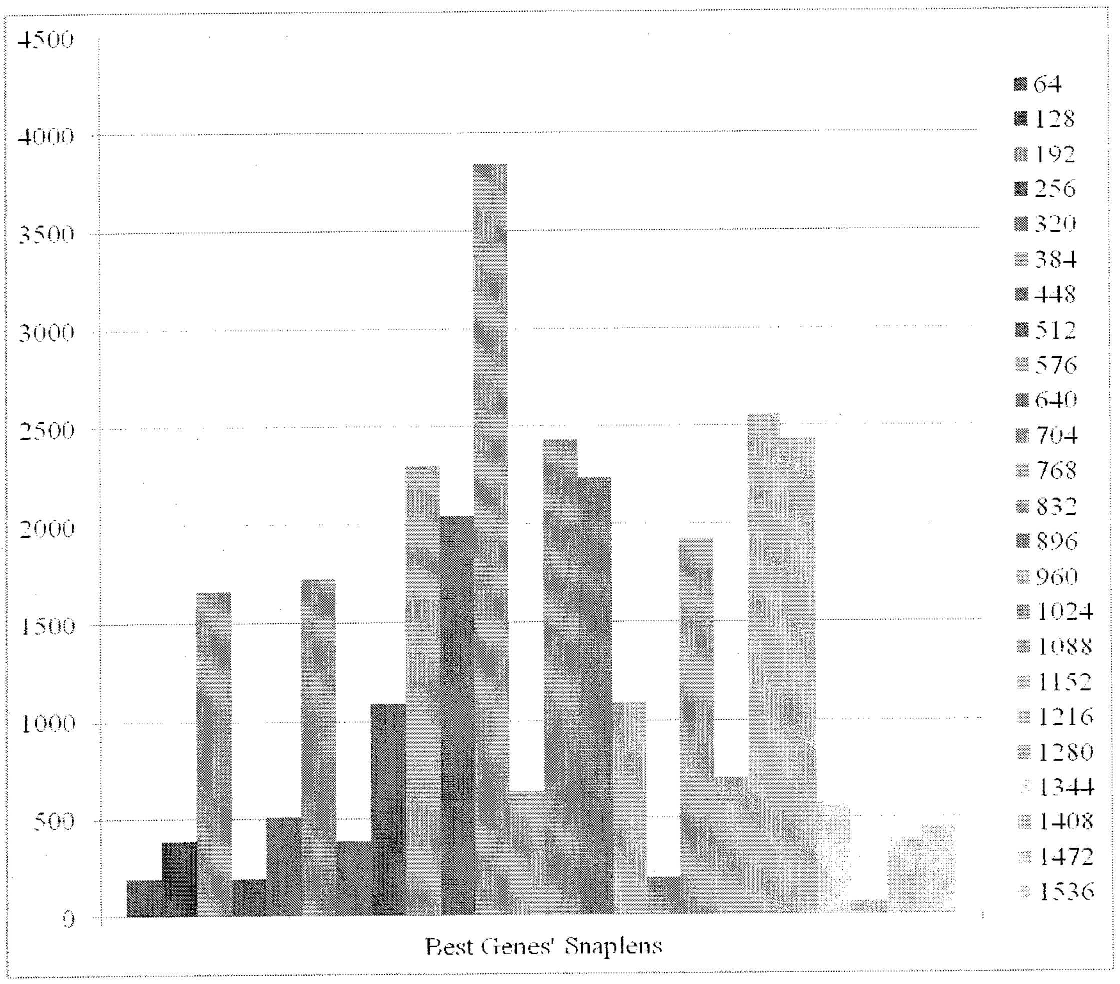


Figure 4.2 Snaplen Values from Different Traffic Loads

Referring to Figure 4.2 and Table 4.1, it is observed that 62.5% of all traffic loads (15 out of 24) have snaplen sizes, which are bigger than their packet size. From here, we can conclude that there is a relation between the size of snaplen and the packet capture rate. The snaplen size must be bigger than the size of each packet of the input traffic, which is logical (snaplen size must accommodate at least one packet). This practice (proper snaplen size) will prevent segmentation that consumes CPU cycle, which eventually makes the network analysis process longer, since the fragmented segments have to be reassembled at the receiving application.

The most significant improvements (compared to previous input traffic's recorded throughput rate) are seen at ping traffics with 768, 960, 1216, 1408, and 1536 bytes of packet size. These particular significant improvements of PF_RING have strengthened the claim by Schneider et al. [38] and Deri [39], who noticed that capturing a 1Gb Ethernet stream with big sized packet was much easier than capturing it with small sized packet (64 bytes).

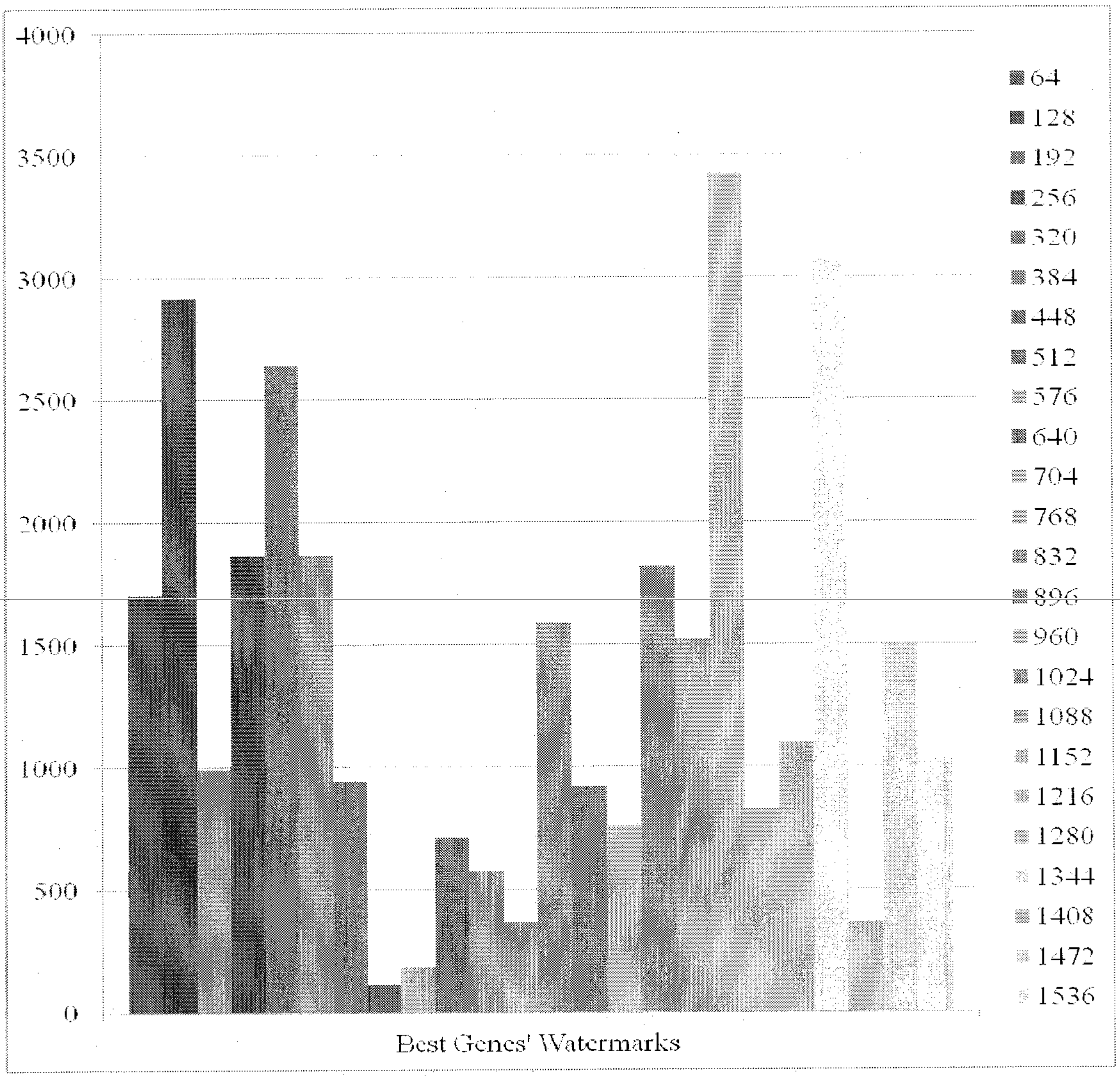


Figure 4.3 Watermark Values from Different Traffic Loads

Figure 4.3 shows the watermark values for all traffic loads. The highest watermark value is 3426 bytes and the lowest is 114 bytes. The 3426 value actually exceeds the specified limit that is 2048 bytes, which was caused by the 7 significant digits of C++ float data type. However, there are only 4 watermark values that exceed 2048 bytes. Analysis from this figure generates 17 out of 24 or 70.83% of all traffic loads have watermark values which are less than half of the highest recorded watermark value. This reflects that the recommended general watermark value is at most half of the specified maximum limit, although PF_RING will still perform well even if it exceeds that half of specified maximum limit. The optimal value of watermark allows the network application not to wait too long (overflown network card capture buffer might occur) before receiving the packets.

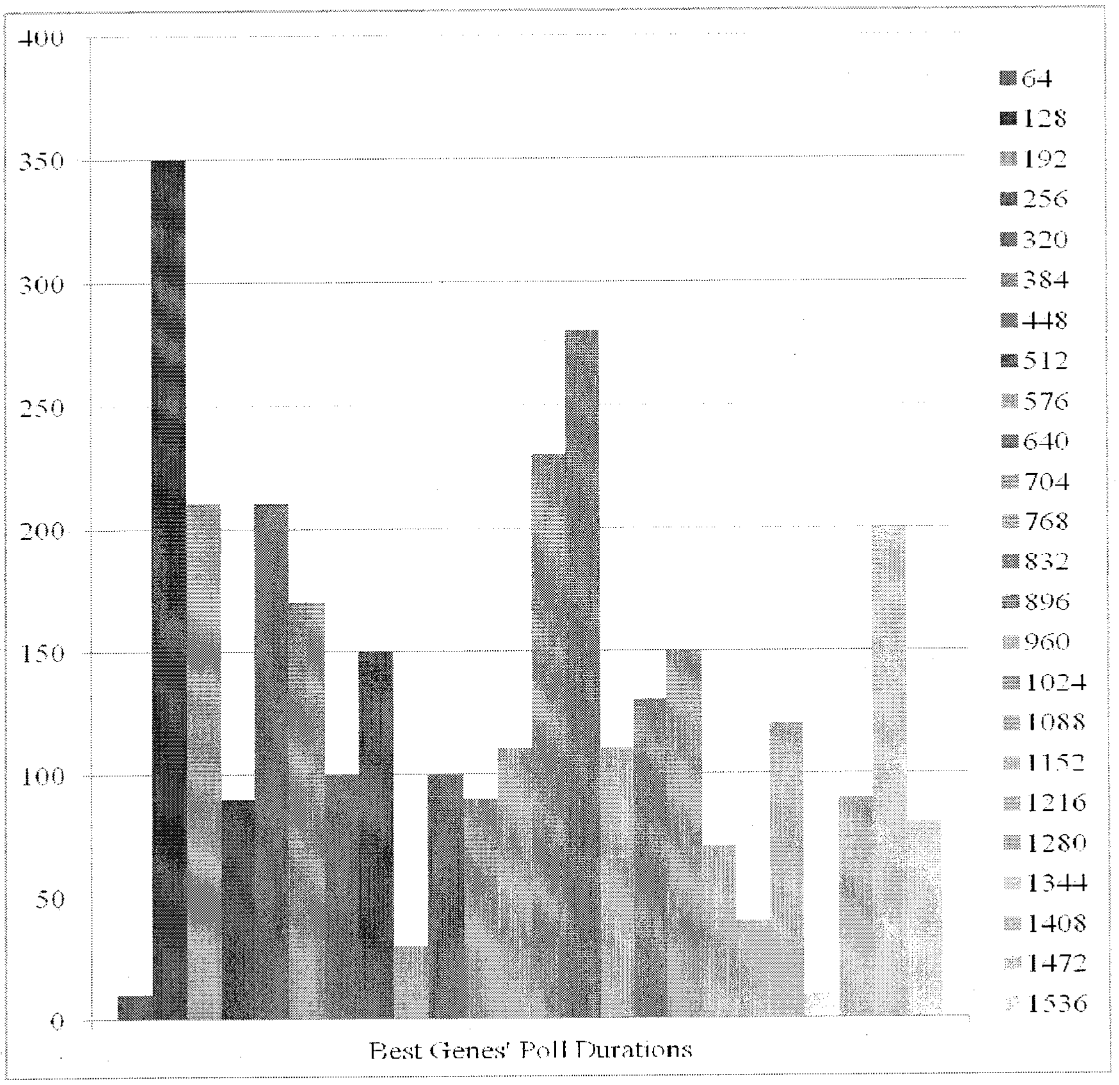


Figure 4.4 Poll Duration Values from Different Traffic Loads

Figure 4.4 displays the poll duration for every traffic load. The analysis was made for three categories. First is for traffic loads that have poll duration equal or below half of highest specified duration (200 ms). Second is for traffic loads that exceed half of highest specified duration (200 ms), and the last is for traffic loads that equal or exceed highest specified duration (200 ms). The results are listed in below table.

Table 4.2 Poll Duration Categories Results

Poll Duration Category	Number of Traffic Load
Duration < = 100ms	11
100ms < Duration < 200ms	7
Duration > = 200ms	6

There are eleven traffic loads (45.83%) in the first category, which are 64, 256, 448, 576, 640, 704, 1152, 1216, 1344, 1408, and 1536 bytes. The second category has seven traffic loads (29.17%), each 384, 512, 768, 960, 1024, 1088, and 1280 bytes. Finally the third category includes 6 traffic loads (25%), namely 128, 192, 320, 832, 896, and 1472. Within all the categories, the traffic loads vary from low/small, moderate/medium, up to high. Additionally, the first category dominates with the ~~most number of traffic loads.~~ From this analysis, it can be interpreted that it is generally suggested to deploy poll duration that is ranging from 10 until 100 ms, for the mentioned traffic load range (64-1536 bytes). 10 until 100ms of poll duration are considered safe and ideal (not too brief, not too long).

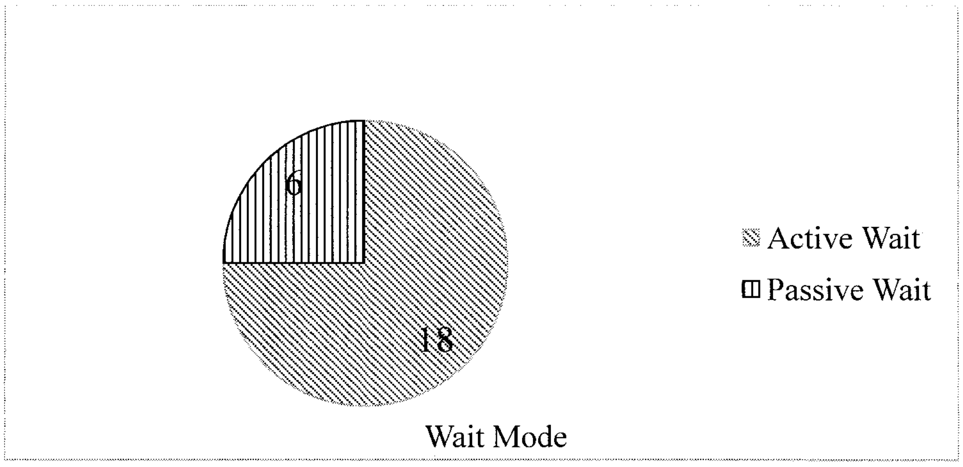


Figure 4.5. The Portion of Wait Mode (Active/Passive) Utilization

Looking at Figure 4.5, it can be seen that 75% or 18 out of 24 traffic loads were treated with Active Wait (no device polling). Based on Table 4.1, for traffic loads of 64-512 bytes, there is only one traffic load (320 bytes) or 12.5% of this range that uses Passive Wait (device polling). It means that within this range, the use of Active

Wait (kernel interrupt for every packet) costs cheaper than device polling. Next, for traffic loads of 576-1024 bytes, there is also only one traffic load (1024 bytes) or 12.5% of this range using Passive Wait. This says that within this range (576-1024 bytes) of traffic load, Active Wait deliberately performs better. The analysis of Wait Mode for traffic load ranging from 1088 to 1536 bytes of packet size shows that there are four traffic loads (1152, 1280, 1344, and 1408 bytes) or 50% of this range that activated Passive Wait. It can be concluded that for this higher load of traffic range, the utilization of device polling delivers high impact (higher throughout rate).

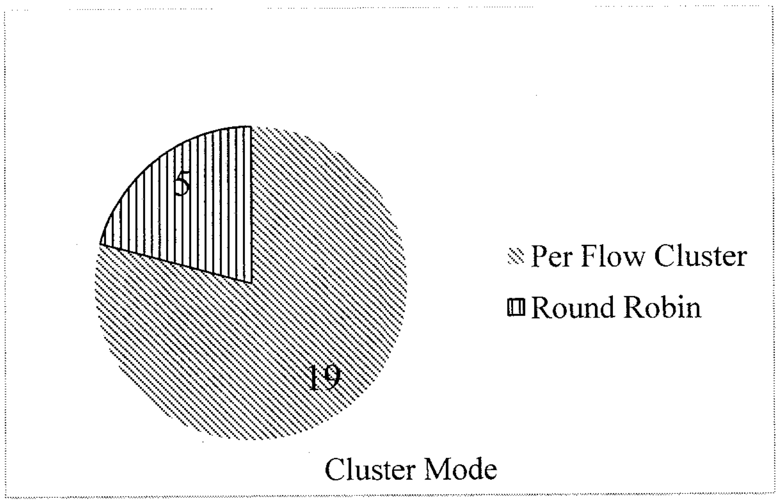


Figure 4.6 Cluster Mode Utilization

Figure 4.6 describes the Cluster Mode utilized by all traffic loads. There are 19 traffic loads (79.17%) utilizing Per Flow cluster, while the rest (5 traffic loads) or (20.83%) use Round Robin cluster. Table 4.1 tells that Round Robin cluster only applies on traffic load of 192, 704, 832, 1408, and 1536 bytes. On the other hand, Per Flow cluster applies on majority of traffic loads ranging from small, medium, to high traffic loads. These findings show that Per Flow cluster gives better packet capturing rate, in general, as compared to Round Robin mode. This is suitable when only one network application takes place and where there is no sharing of packets among multiple applications. Therefore the single application will only take care on its own buffer, and it will get all parts of a transmission flow without the need to reassemble if those parts are separated.

Table 4.1 also depicts the contents of the selected best chromosome of PF_RING configuration and its firewall rules ordering for 24 different traffic loads (64-1536 bytes). There are 13 rules excluding the default 'deny' rule at the end of the set, which is not included in reordering process. Analysis from the firewall rules ordering column of Table 4.1, it can be observed that 14 out of 24 chromosomes (58.33%) put rule no. 13, which has 'accept' action as the front line of the rule set. The chromosomes with this characteristic are for traffic load of 64, 256, 384, 448, 512, 576, 704, 768, 896, 1024, 1088, 1152, 1280, and 1344 bytes for each packet size. The traffic loads with this anomaly range from small up to high traffic loads. Thus, it can be concluded that our genetic algorithm approach works well in ordering or reordering the firewall rules. Especially where the rule set tends to have more rules to accept packets than to drop them. This is useful when the new administrator is facing the challenge of inserting and reordering new rules into the existing rule set.

4.2.2. Results and Discussion of the Second Experiment

The second experiment was conducted to justify the findings in the first experiment; we increase the number of population, the number of generation, the probability of mutation, and the probability of crossover to see if the patterns change, which might show the unjustified results from the first experiment. The tables and graphs that depict the results of the second experiment are shown below.

Table 4.3 The Chosen Best Chromosome Contents for Different Traffic Load Levels

No.	Packet Size (bytes)	Best Chromosome Throughput (bytes/7 sec)	Snaplen (bytes)	Active/ Passive Wait	Watermark Value (bytes)	Cluster Type	Poll Duration (ms)	Firewall Rules Order
1.	64	4397.64	768	Active wait	829	Round robin	310	11-4-1-10-9-7-13-5-6-8-2-3-12
2.	128	6627.26	128	Active wait	2276	Round robin	60	11-8-5-12-4-13-10-1-3-6-7-9-2
3.	192	8120	1856	Active wait	1701	Per flow	50	6-4-12-7-2-13-5-8-11-1-3-10-9
4.	256	10181.1	640	Active	1	Per flow	110	3-2-13-

				wait				10-5-1-6-8-7-11-12-9-4
5.	320	11559.2	2048	Passive wait	24	Per flow	160	5-13-7-8-3-4-2-9-6-11-12-10-1
6.	384	13164.3	192	Active wait	1425	Per flow	150	8-3-11-2-12-13-9-10-7-5-1-4-6
7.	448	16779	2368	Active wait	2071	Per flow	150	3-11-6-1-13-7-8-4-5-2-9-12-10
8.	512	17157.8	2752	Passive wait	4371	Per flow	190	4-7-2-11-13-5-3-10-8-12-6-9-1
9.	576	18991.1	896	Active wait	24	Per flow	150	7-3-5-13-12-1-6-10-11-9-4-2-8
10.	640	20667.4	192	Active wait	5105	Per flow	380	3-7-11-13-1-4-12-6-10-8-9-5-2
11.	704	23455.3	2304	Active wait	2000	Per flow	130	7-10-3-11-6-8-2-4-1-13-9-5-12
12.	768	24655.4	64	Active wait	1402	Round robin	30	3-1-2-5-4-12-10-6-11-13-7-9-8
13.	832	26637.7	704	Active wait	1496	Per flow	270	4-1-10-13-11-3-5-12-9-2-7-6-8
14.	896	30003.3	576	Active wait	1954	Per flow	500	8-3-10-6-2-11-5-12-13-1-9-4-7
15.	960	30642.9	1728	Active wait	1264	Per flow	80	6-10-13-12-7-9-5-11-4-2-8-3-1
16.	1024	32990.2	2112	Active wait	1126	Per flow	210	5-7-3-12-9-10-2-1-4-8-6-11-13
17.	1088	34105	1984	Active wait	275	Per flow	120	10-12-7-4-3-8-2-11-1-13-9-6-5
18.	1152	36118.6	2496	Passive wait	390	Round robin	180	11-9-8-13-5-1-2-6-7-12-3-10-4
19.	1216	38321.9	3200	Passive wait	1542	Per flow	120	10-4-5-6-9-13-3-2-8-1-11-7-12
20.	1280	40026.2	1472	Active	1448	Round	370	7-10-8-9-

				wait		robin		12-4-6-13-3-1-11-5-2
21.	1344	43994.8	640	Active wait	24	Round robin	240	1-9-3-13-12-7-8-4-6-2-11-5-10
22.	1408	44122.3	768	Passive wait	461	Per flow	30	4-7-2-12-9-6-10-8-11-1-13-5-3
23.	1472	45288	640	Passive wait	2506	Round robin	10	4-10-8-11-12-2-7-5-13-1-9-6-3
24.	1536	49178.6	128	Active wait	760	Per flow	270	5-11-13-1-3-12-7-10-8-4-6-2-9

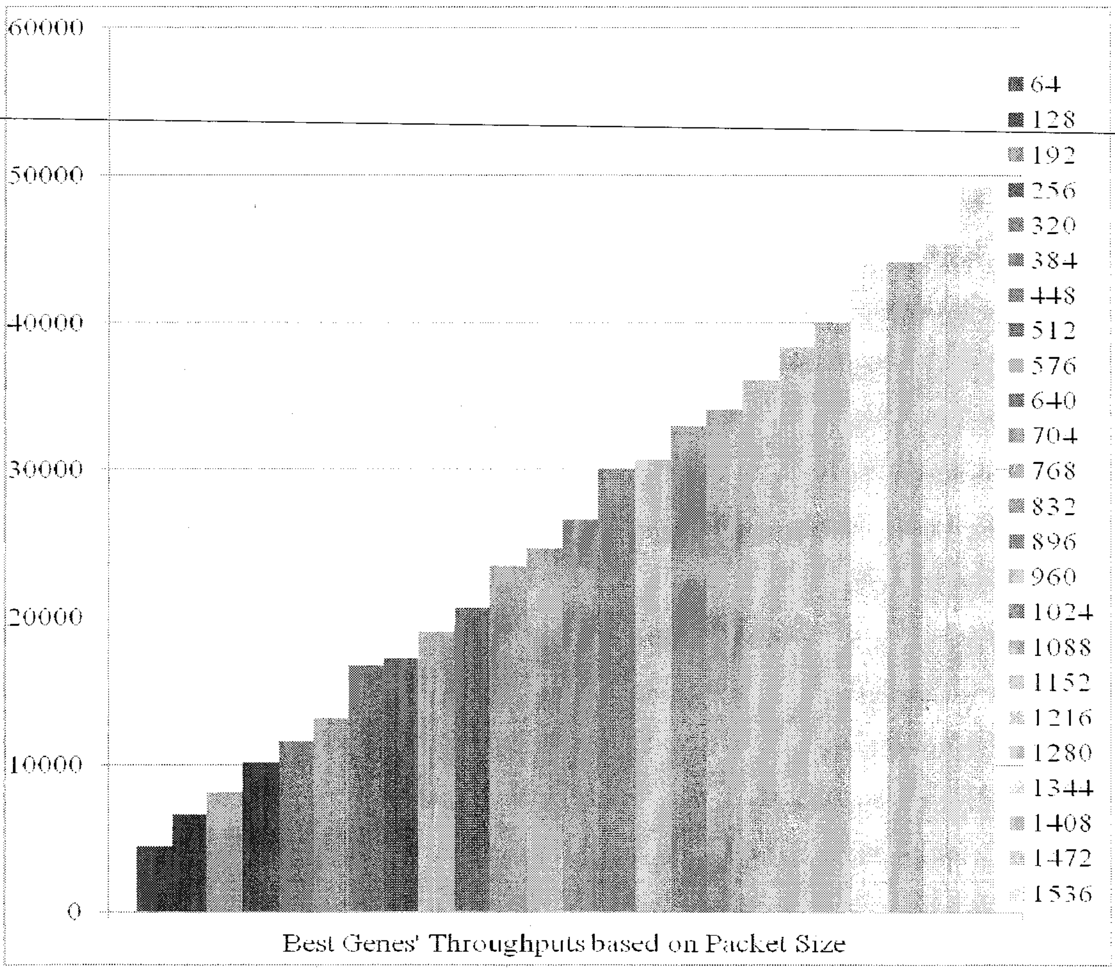


Figure 4.7 Throughput Progression from Different Traffic Loads

Table 4.3 and Figure 4.7 show the similar increasing pattern as in Table 4.1 and in Figure 4.1. This justified the findings in the initial experiment, because there is no change of fitness pattern. It reflects that our first experiment is sufficient with 50 populations, 50 generations, 0.2 of mutation probability, and 0.4 of crossover probability.

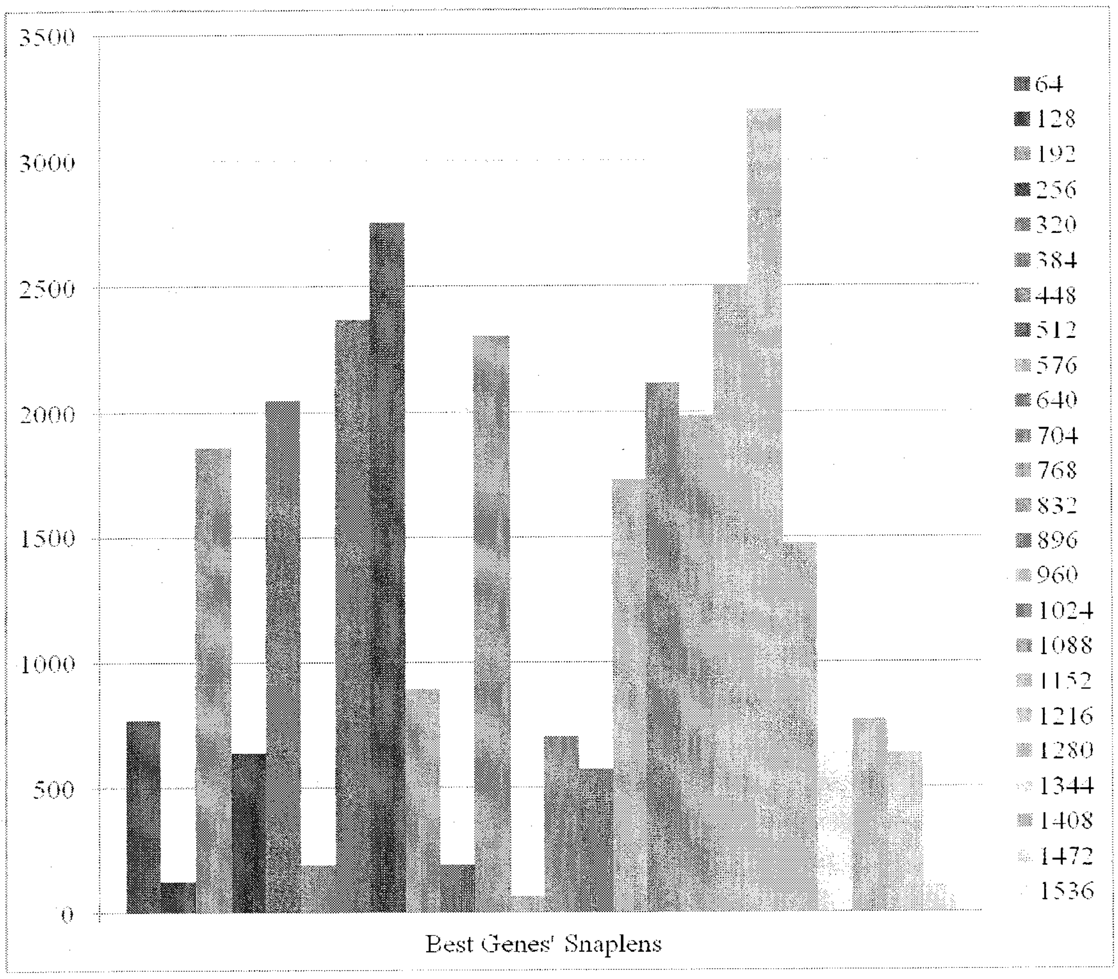


Figure 4.8 Snaplen Values from Different Traffic Loads

From Figure 4.8 and Table 4.3, they show exactly the same amount of traffic loads with snaplen size bigger than their packet size as in the first experiment, that is, 62.5% of all traffic loads or 15 out of 24 traffics. This again justifies the results and analysis from the first experiment.

Again, it is found that the most significant improvements are in ping traffic with big sized packet (448, 704, 896, 1024, 1344, and 1536 bytes). This conforms to the findings in the initial experiment.

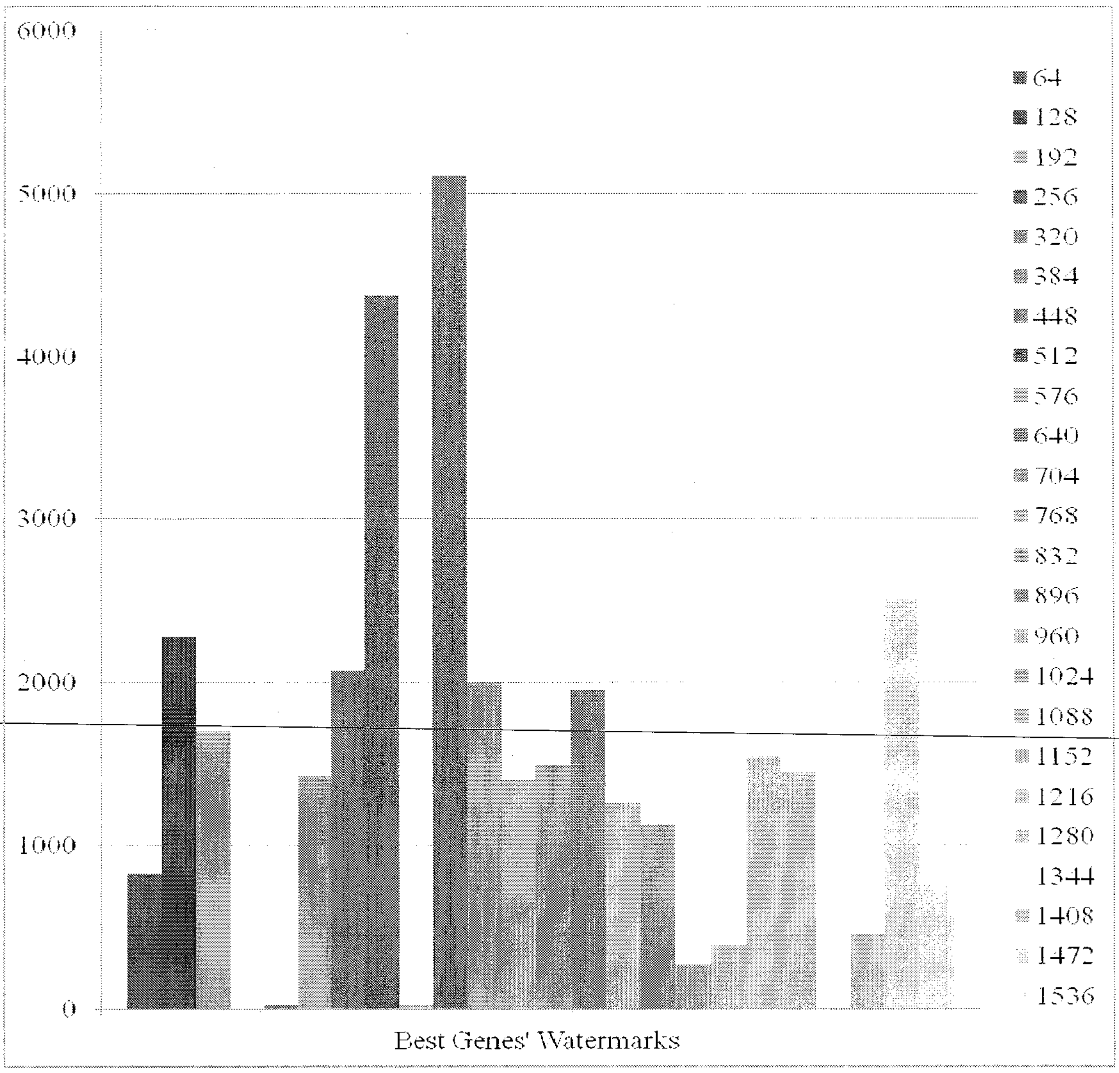


Figure 4.9 Watermark Values from Different Traffic Loads

In Figure 4.9, the highest watermark value is 5105 bytes and the lowest is 1 byte. The 5105 value goes beyond the specified limit of 2048 bytes caused by the 7 significant digits of C++ float data type. But, there are only 5 watermark values that exceed 2048 bytes. These are almost the same as in the first experiment that has 4 of them. Figure 4.9 shows 22 out of 24 or 91.67% of traffic loads have watermark values less than half of the highest recorded watermark value (5105). This also strengthens the analysis of watermark values from the first experiment.

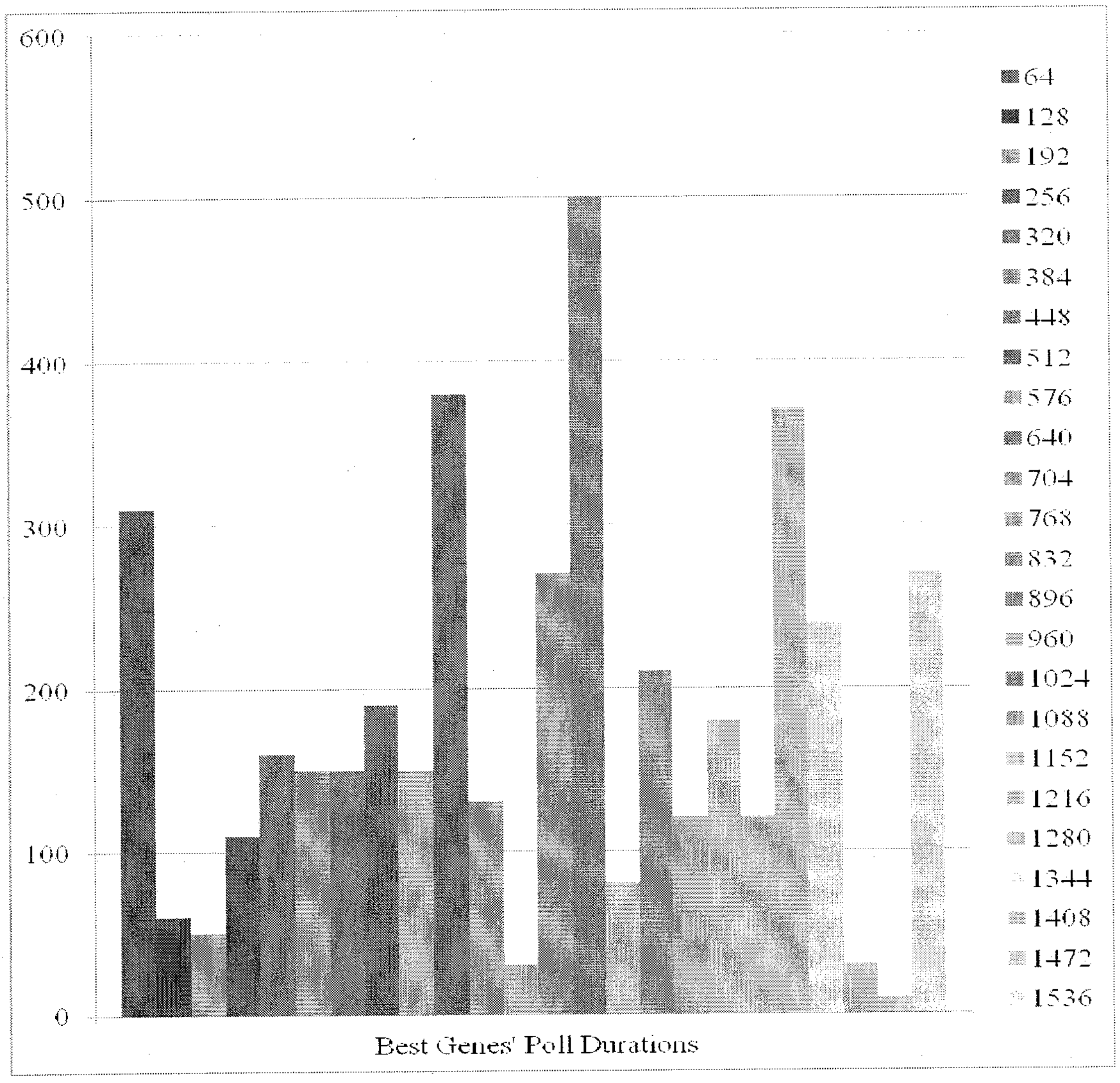


Figure 4.10 Poll Duration Values from Different Traffic Loads

Figure 4.10 displays the poll duration for every traffic load. As in the first experiment, the analysis is based on three categories as depicted in below table.

Table 4.4 Poll Duration Results

Poll Duration Category	**Number of Traffic Load**
Duration < = 100ms	6
100ms < Duration < 200ms	10
Duration > = 200ms	8

Depicted by Table 4.4, there are 6 traffic loads (25%) in the first category, which include 128, 192, 768, 960, 1408, and 1472 bytes. The second category consists of 10 traffic loads (41.67%), which are 256, 320, 384, 448, 512, 576, 704, 1088, 1152, and 1216 bytes. The third category has 8 traffic loads (33.33%), each 64, 640, 832, 896, 1024, 1280, 1344, and 1536. This experiment shows the domination of the second category. It is different from the first experiment, hence it can be concluded that the setting of poll duration is effective and stable in the range of 10 until less than 200 ms, for the experimented traffic load range (64-1536 bytes).

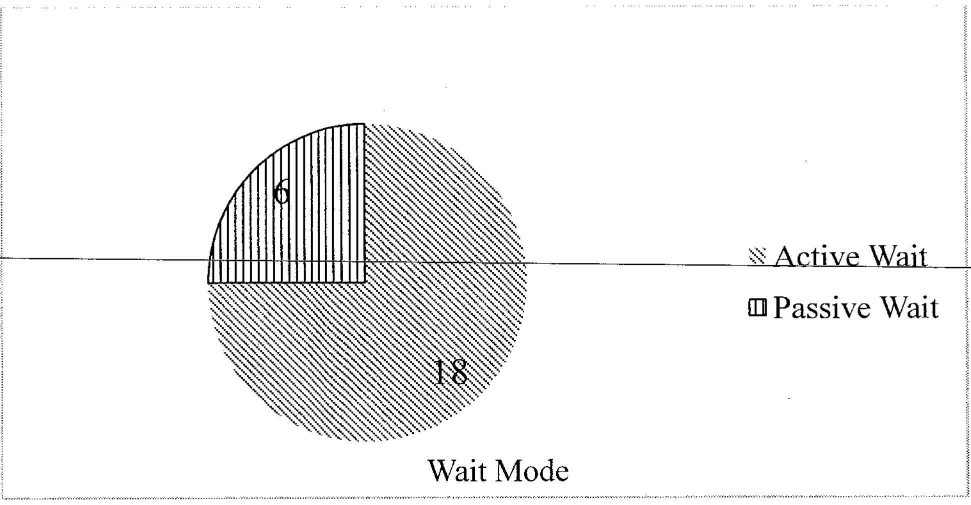


Figure 4.11. The Portion of Wait Mode (Active/Passive) Utilization

Referring to Figure 4.11, it shows that 75% or 18 out of 24 traffic loads were treated with Active Wait (no device polling). In Table 4.3, for range of traffic loads of 64-512 bytes, there are 6 traffic loads with Active Wait (75%), which means that within this range, the use of Active Wait is better. This conclusion is the same as in the first experiment. For traffic loads from 576 until 1024 bytes, all of the traffic loads utilize Active Wait. This also supports the analysis in the first experiment. At traffic loads ranging from 1088 to 1536 bytes of packet size, there are 4 traffic loads (50%) with Active Wait and 4 traffic loads (50%) with Passive, which show the same statistics as in the first experiment. This infers that for higher load of traffic, the need of device polling is becoming important.

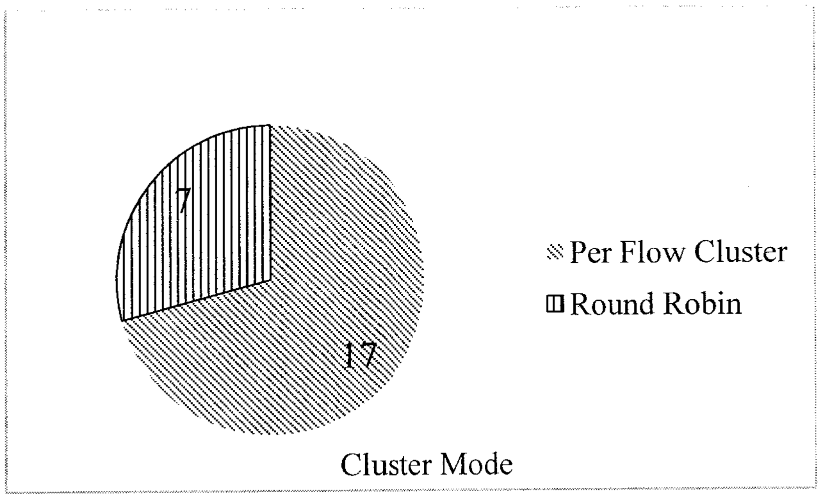


Figure 4.12 Cluster Mode Utilization

Based on Figure 4.12, there are 17 traffic loads (70.83%) deploying Per Flow cluster, while the other 7 (29.17%) utilize Round Robin cluster, which occurs at traffic loads with 64, 128, 768, 1152, 1280, 1344, and 1472 bytes of packet size. This finding also conforms to the finding in the first experiment, where Per Flow cluster mode drives better packet capturing rate in average against Round Robin mode.

The part about firewall rules reordering in the second experiment is shown in Table 4.3. It can be seen that 16 out of 24 chromosomes (66.66%) put rule no. 13 with 'accept' action at the front line of the rule set. Those chromosomes are for traffic load 64, 128, 192, 256, 320, 384, 448, 512, 576, 640, 832, 960, 1152, 1216, 1344, and 1536 bytes of each packet size. The traffic loads with this property cover small up to high traffic loads. This result is slightly increased from the first experiment (which has 14 chromosomes) with 'accept' action at the front line.

Overall, the results of the first and the second experiment are almost identical. Therefore, it is concluded that the first experiment with genetic algorithm properties of 50 generations, 50 populations, mutation probability of 0.2, and crossover probability of 0.4 is sufficient to generate optimized packet filtering solution.

4.3. Summary

This chapter covers the results and findings of this research's experiments. The experiment was conducted twice with different genetic algorithm properties to justify the findings. Results were analyzed to find the specific patterns which were related to PF_RING kernel parameters and firewall rules ordering. These patterns gave clue of how those kernel parameters were adjusted in specific situation (different loads of input traffic). The firewall rules ordering were observed to evaluate its effectiveness. The analysis of two experiments showed that the objective to optimize packet filtering using genetic algorithm method was achieved.

CHAPTER 5

CONCLUSIONS, CONTRIBUTIONS, AND FUTURE WORK

5.1. Conclusions

Genetic algorithm methodology can optimize both PF_RING kernel parameters and firewall rules ordering, in order to make up an optimized packet filtering framework. The GAlite program in this research can perform the genetic algorithm process, based on the packet capture kernel module (PF_RING) kernel parameters and its firewall rules. The results and their analysis show that the proposed method is suitable for different levels of traffic load. Different input load requires particular packet capture setting of kernel parameters. However, the firewall rules reordering does not depend on the size of traffic load. Proper rules ordering maximizes the throughput rate regardless of the size of traffic load.

This optimization framework will assist an administrator in setting the optimal packet filter or merely packet capture configuration when specific network packet transmission, for example a file transfer is about to occur, especially when the traffic input rate can be estimated.

Finally, the overall network analysis process and objectives as mentioned in Chapter 2 can be accelerated with optimized packet capturing.

5.2. Contributions

This research's contributions lie at the optimization of network packet filtering using genetic algorithm methodology. The accomplished optimized packet filtering itself consists of two processes, which are packet capturing and packet filtering using firewall rules. The contribution towards packet capturing is located at the optimization of PF_RING kernel parameters for different input traffic loads (different load requires different treatment). Optimized packet capturing will automatically accelerate the packet filtering phase which in this case is using firewall rules. Another contribution is made in packet filtering scope by optimizing the firewall rules ordering. The optimization of firewall rules ordering relates to the automatic reordering of rules, especially when new rules are to be inserted into the existing rule set. Both the optimized kernel parameters and firewall rules ordering deliver a whole optimized packet filtering framework.

Overall, this research patches the previous packet capturing methods that do not adjust (optimized) to specific input traffic loads. Furthermore, this research also adds acceleration to previous packet filtering methods which only focus on the reordering of firewall rules without considering the optimized acceleration of packet capturing beneath it.

This research's constraint is the isolation of the specified load of input traffic when we run the experiment, in order to find the optimal packet filtering configuration. To conduct an accurate experiment, the measured input traffic is preferred to be sourced from local network and it must be isolated and not be mixed with other network stream e. g. Internet traffic. The failure to isolate input traffic will increase the intended size of input traffic load, because the Internet packets will add up to the stream. On the other hand, the isolation of input traffic might cause an Internet service to be unavailable, hence it is suggested to conduct the experiment at the hour where the Internet service is least accessed, for example: at midnight.

5.3. Future Work

In the future, our framework can be applied in other prospective open source packet capturing and/or filtering module. Thus, it can become a generic packet filtering optimization method. Energy measurement can also be added to develop a green packet filtering framework based on the power consumption for specific configuration of packet filtering parameters. The filtering technique can also be modified. Another filtering technique other than firewall rules is for instance: BPF. Furthermore, a change in PF_RING transparent mode is worth trying as well. This research can also be deeper explored using higher performance network card, for example is using network card with 10Gbps speed. Moreover, this research may be repeated under different variances of operating system e. g. Red Hat based Linux distributions, FreeBSD, or even Sun Solaris.

REFERENCES

[1] G. Held, "The TCP/IP Protocol Suite," in *Ethernet Networks*, 4th ed. West Sussex, UK: Wiley, 2003, ch. 5, sec. 4, pp. 244-277.

[2] S. Panwar *et al.*, "TCP/IP Overview," in *TCP/IP Essentials: A Lab-Based Approach.* Cambridge, UK: Cambridge University Press, 2004, ch. 0, sec. 2, pp. 2-8.

[3] T. Lammle and A. Barkl, "Network Protocols," in *CCDA: Cisco Certified Design Associate Study Guide,* 2nd ed. California: Sybex, 2003, ch. 3, sec. 1, pp. 120-131.

[4] E. Coll, (2008, April 12). *Fundamentals of Datacom and Networking* [Video]. Available: http:// www.teracomtraining.com

[5] E. Coll (2008, April 12). *Understanding Networking 1* [Video]. Available: http:// www.teracomtraining.com

[6] E. Coll (2008, April 12). *Understanding Networking 2* [Video]. Available: http:// www.teracomtraining.com

[7] J. E. Canavan, "Threats, Vulnerabilities, and Attacks," in *Fundamentals of Network Security,* Norwood: Artech House, 2001, ch. 2, pp. 25-47.

[8] J. A. Vacca, "Preventing System Intrusions," in *Network and System Security.* Burlington: Syngress, 2010, ch. 3, pp. 60-81.

[9] A. B. Aissa, R K. Abercrombie, F. T. Sheldon and A. Mili. (2010, Mar.). Quantifying security threats and their potential impacts: a case study. *Innovations in Systems and Software Engineering* [Paper]. *6(4)*, pp. 269-281.

[10] B. Dunsmore *et al,* "Securing Your Internetwork," in *Mission Critical Internet Security.* Rockland: Syngress, 2001, ch. 1, pp. 6-40.

[11] B. Toxen, "Common Hacker Attacks," in *Real World Linux© Security: Intrusion Prevention, Detection, and Recovery,* 2nd ed. New Jersey: Prentice Hall, 2002, ch. 5, pp. 236-257.

[12] E. Maiwald, "Types of Attacks," in *Network Security: a Beginner's Guide*. New York: McGraw-Hill, 2001, ch. 2, pp. 15-26.

[13] S. Garfinkel *et al,* "TCP/IP Networks," in *Practical Unix & Internet Security*, 3rd ed. Sebastopol: O'Reilly, 2003, ch. 11.

[14] A. Lockhart, "Network Security," in *Network Security Hacks*, 2nd ed. Sebastopol: O'Reilly, 2006, ch. 6.

[15] L. Gheorghe, "Security Threats," in *Designing and Implementing Linux Firewalls and QoS using netfilter, iproute2, NAT, and L7-filter.* Birmingham, UK: Packt Publishing, 2006, ch. 2, pp. 42-55.

[16] E. D. Zwicky, S. Cooper and D. B. Chapman, "Why Internet Firewalls?, " in *Building Internet Firewalls*, 2nd ed. Sebastopol: O'Reilly, 2000, ch. 1, pp. 9-27.

[17] L. Bernstein (2007, Oct.). Network Management Isn't Dying, It's Just Fading Away. *Journal of Network and Systems Management* [Paper]. *15(4)*, pp. 419-424.

[18] R. L. Ziegler, "Packet-Filtering Concepts," in *Linux Firewalls,* 2nd ed. Indianapolis: New Riders, 2001, ch. 2, pp. 27-79.

[19] T. W. Ogletree, "Packet Filtering," in *Practical Firewalls,* 1st ed. Que, 2000, ch. 5, pp. 65-76.

[20] J. Govaerts, A. Bandara and K. Curran. (2009, Nov.). A formal logic approach to firewall packet filtering analysis and generation. *Artificial Intelligence Review* [Paper]. *29(3-4)*, pp. 223-248.

[21] J. G. Alfaro, N. Boulahia-Cuppens and F. Cuppens. (2007, Oct.). Complete analysis of configuration rules to guarantee reliable network security policies. *International Journal of Information Security* [Paper]. *7(2),* pp. 103-122.

[22] E. S. Al-Shaer and H. H. Hamed (2004, Mar.). Discovery of policy anomalies in distributed firewalls. *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies. 4*, pp. 2605-2616.

[23] E. S. Al-Shaer and H. H. Hamed (2002). Design and Implementation of Firewall Policy Advisor Tools. *Technical Report CTI-techrep0801*. pp. 1-21.

[24] M. Sourour, B. Adel and A. Tarek. (2009, Mar.). Ensuring security in depth based on heterogeneous network security technologies. *International Journal of Information Security* [Paper]. *8(4)*, pp. 233-246.

[25] A. Prathapani, L. Santhanam and D. P. Agrawal. (2011, Jan.). Detection of blackhole attack in a Wireless Mesh Network using intelligent honeypot agents. *The Journal of Supercomputing* [Online]. Available: http://www.springerlink.com/content/t0w30t95j0901234/fulltext.pdf

[26] L. Z. Cai, J. Chen, Y. Ke, T. Chen and Z. G. Li. (2010, Jan.). A new data normalization method for unsupervised anomaly intrusion detection. *Journal of Zhejiang University-SCIENCE C (Computers & Electronics)* [Paper]. *11(10)*, pp. 778-784.

[27] M. Sheikhan, Z. Jadidi and A. Farrokhi. (2010, Nov.). Intrusion detection using reduced-size RNN based on feature grouping. *Neural Computing & Applications* [Online]. Available: http://www.springerlink.com/content/b66603nk1x8373h68/fulltext.pdf

[28] H. Jin, G. Xiang, D. Zou, S. Wu, F. Zhao, M. Li and W. Zheng. (2011, Apr.). A VMM-based intrusion prevention system in cloud computing environment. *The Journal of Supercomputing* [Online]. Available: http://www.springerlink.com/content/633jq1233k94pnu6/fulltext.pdf

[29] M. Zulkernine, M. Graves and M. U. A. Khan. (2007, May). Integrating software specifications into intrusion detection. *International Journal of Information Security* [Paper]. *6(5)*, pp. 345-357.

[30] G. Folino, C. Pizzuti, and G. Spezzano. (2010, Feb.). An ensemble-based evolutionary framework for coping with distributed intrusion detection. *Genetic Programming and Evolvable Machines* [Paper]. *11(2)*, pp. 131-146.

[31] M. Naveed, S. Nihar and M. I. Babar. (2010, Nov.). Network Intrusion Prevention by Configuring ACLs on the Routers, based on Snort IDS alerts. *Proceedings of the 2010 6th International Conference on Emerging Technologies (ICET)* [Paper]. pp. 234-239.

[32] SonicWALL. *SonicOS Enhanced 4.0: Packet Capture.*

[33] Wireshark Purposes. http://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html#ChIntroPurposes

[34] Inline Snort Final. http://openmaniak.com/inline_final.php

[35] Inline Snort. http://openmaniak.com/inline.php

[36] L. Ricciulli and T. Covel (2011, Sept.). *Inline Snort multiprocessing with PF_RING.* Snort Setup Guides.

[37] L. Braun, A. Didebulidze, N. Kammenhuber and G. Carle (2010, Nov.). Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware. *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement.* pp. 206-217.

[38] F. Schneider, J. Wallerich and A. Feldmann (2007, Apr.). Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. *Proceedings of the 8th International Conference on Passive and Active Network Measurement.*

[39] L. Deri (2004, Oct). Improving Passive Packet Capture: Beyond Device Polling. *Proceedings of the 2004 4th International System Administration and Network Engineering Conference* [Paper]. pp. 1-10.

[40] M. Dashtbozorgi and M. A. Azgomi (2010, Aug.). A high-performance and scalable multi-core aware software solution for network monitoring. *The Journal of Supercomputing* [Paper]. *59(2)*, pp. 720-743.

[41] J. Mogul and K. K. Ramakrishnan (1997, Aug.). Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems. 15(3),* pp. 217–252.

[42] I. Kim, J. Moon and H. Y. Yeom (2001, Sept.). Timer-based interrupt mitigation for high performance packet processing. *Proceedings of 5th International Conference on High-Performance Computing in the Asia-Pacific Region.*

[43] L. Rizzo (2001, Nov.). Device Polling Support for FreeBSD. *Proceedings of BSDCon Europe Conference.*

[44] G. A. Cascallana and E. M. Lizarrondo (2006, Sept.). Collecting packet traces at high speed. *Proceedings of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2006.*

[45] L. Deri and F. Fusco (2009, July). Exploiting commodity multi-core systems for network traffic analysis. Available: http://luca.ntop.org/MulticorePacketCapture.pdf

[46] F. Schneider and J. Wallerich (2005, Oct.). Performance evaluation of packet capturing systems for high-speed networks. *Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. pp. 284–285.

[47] L. Deri (2007, June). High-Speed Dynamic Packet Filtering. *Journal of Network and Systems Management* [Paper]. *15(3)*, pp. 401-415.

[48] DMA Ring. http://www.cs.ucla.edu/~kohler/class/05s-osp/notes/notes12.html

[49] T. Okamoto (2011, Feb.). An artificial intelligence membrane to detect network intrusion. *Proceedings of the 15th International Symposium on Artificial Life and Robotics* [Paper]. *16*, pp. 44-47.

[50] K. Sastry, D. Goldberg and G. Kendall, "Genetic Algorithms", in *Search Methodologies*, Edited E. K. Burke and G. Kendall, Los Angeles: Springer US, 2005, ch. 4, pp.97-125

[51] A. Shrivastava and S. Hardikar (2012, July). Performance Evaluation of BPNN and Genetic Algorithm. *VSRD International Journal of CS & IT* [Paper]. *2(7)*, pp. 621-628.

[52] E. El-Alfy (2007, Feb.). A Heuristic Approach for Firewall Policy Optimization. *The 9th International Conference on Advanced Communication Technology* [Paper]. *3*, pp. 1782–1787.

[53] A. T. Nottingham and B. Irwin (2009, Aug.). gPF: A GPU Accelerated Packet Classification Tool. *Southern Africa Telecommunication Networks and Applications Conference* [Online]. Available: http://www.satnac.org.za/proceedings/2009/papers/software/Paper%2063.pdf

[54] A. S. Tongaonkar (2004, May). Fast Pattern-Matching Techniques for Packet Filtering. *Stony Brook University Tech. rep.*

[55] A. Nottingham and B. Irwin (2009, July). Investigating the Effect of Genetic Algorithms on Filter Optimisation within Fast Packet Classifiers. *Proceedings of the ISSA 2009 Conference*. pp. 99-116.

[56] P. T. Chen and C. S. Laih (2007, June). IDSIC: an intrusion detection system with identification capability. *International Journal of Information Security* [Paper]. *7(3),* pp. 185-197.

[57] GAlib. http://lancet.mit.edu/ga/

[58] GAlite. http://code.google.com/p/galite/

[59] TNAPI. http://www.ntop.org/products/pf_ring/tnapi/

[60] B. H. Leitao (2009, July). Tuning 10Gb network cards on Linux: A basic introduction to concepts used to tune fast network cards. *Proceedings of the Linux Symposium* [Paper]. pp. 169-184.

[61] NTOP PF_RING. http://www.ntop.org/products/pf_ring/

[62] PF_RING Transparent Mode. http://www.ntop.org/pf_ring/pf_ring-and-transparent-mode/

[63] PF_RING DNA. http://www.ntop.org/products/pf_ring/dna/

[64] RSS. http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm

[65] NTOP (2012, June). PF_RING User Guide. *Linux High Speed Packet Capture.*

[66] C. Dovrolis, B. Thayer and P. Ramanathan (2001, Oct.). HIP: Hybrid Interrupt-Polling for the Network Interface. *Proceedings of 2001 ACM SIGOPS Operating Systems Review. 35(4),* pp. 50-60.

[67] Linux Hardware Interrupt. http://www.linux-tutorial.info/modules.php?name=MContent&pageid=86

[68] Timing in the Linux Kernel. http://www.6test.edu.cn/~lujx/linux_networking/0131777203_ch02lev1sec7.html

PUBLICATIONS LIST

O. Nurika, A. Hudaya, A. Sani and N. Zakaria, "Review of Various Firewall Deployment Models," in *International Conference on Computer and Information Sciences 2012 (ICCIS2012)*, Kuala Lumpur, 2012, pp. 825-829.

O. Nurika, N. Zakaria and L. T. Jung, "Genetic Algorithm Optimized Packet Filtering," *International Conference on Convergence and its Application (ICCA 2013)*, Seoul, Apr. 2013. (Under Review)