A Performance Study of the Snort IDS

Eric Frimpong
M.H. MacGregor

TR08-04
Department of Computing Science
University of Alberta
February, 2008

1.0 INTRODUCTION

With the enormous growth of the IP network, network security has become one of the important issues facing corporations today. It is important that a business is able to protect the private data of its customers and business strategies from unauthorized access. Failure to do this can greatly tarnish the reputation of the business in question and can also lead to high financial losses. Hence it becomes the responsibility of corporations to ensure data confidentiality, integrity and source authentication of their systems. It is important that they are notified in real-time of any malicious unauthorized access to their system. So how can a corporation protect its network from an outside intrusion effectively? The simple answer to this question is by deploying an effective Intrusion Detection System (IDS). According to Wikipedia, intrusion detection is the act of detecting actions attempt to compromise the confidentiality, integrity or availability of a resource and the system that performs these activities automatically is known an IDS. [1]

There are many network intrusion detection and prevention tools available to the network administrator of today, but Snort has become an enterprise standard due to its open source nature and also due to the fact that there are many open source operating systems which work very well with Snort. It is therefore very important for network and system administrators to know and understand the strengths and weaknesses of this tool before deploying it on their system. This project sets out to provide some a type of reference model whereby the performance of Snort can be measured under certain "defined" conditions. Snort can be used as a packet sniffer, packet logger or a network intrusion detection system but in this experiment, the use of snort will be mainly as a network intrusion detection system. In particular, investigating two behaviors that are attributed to Snort, namely:

- Under high load conditions, Snort drops packets without informing the network administrator and
- Under high load conditions, Snort allows packets to pass which violates one or more rules in the rule set.

2.0 Background Information on Snort

Snort is increasingly becoming one of the most deployed IDS in the networking industry and as a result many publications have been made on it. In the first part, a general overview of Snort will be presented based on surveys of previous journal and conference publications. This will be followed by a second part focusing on the inner workings of Snort.

2.1 Survey on Snort

Due to its wide deployment in IP networks, work conducted on Snort has been focused on improving its performance. This enhancement in performance is illustrated by the following publications.

2.1.1 WIND: Workload-Aware Intrusion Detection presented by *Sushant Sinha, Farnam Jahanian, and Jignesh M. Patel.* [2]

One of the challenges facing Snort is its rule matching ability; how fast and efficient it is able to match rules of increasing in number and in rule complexity. To help in this direction the authors of this paper base their work on the premise that: to get a high performance on any IDS, it should be able to adapt accordingly to the workload it encounters which includes the rule set and the network traffic characteristics. To achieve this goal they have developed an adaptive algorithm that can systematically inspect the network traffic and the rule set supplied to the IDS to come out with a "high performance and memory-efficient packet inspection strategy". In this regard they have developed two distinct components over Snort: a profiler which is responsible for analyzing the rule set and the network traffic to come out with a rule matching strategy; and an evaluation engine which is responsible for pre-processing the rule set according to the strategy that was developed by the profiler and then subsequently evaluate the incoming packets to determine which rules can be applied to the packets.

2.1.1.1 Summary of Implementation Details

For the IDS to be able to adapt dynamically to workload and make use of memory and CPU cycles efficiently, the authors of this paper proposed separating the rules into groups by using the protocol field in a given rule and the rule groups are chosen to be kept in memory based on the idea that "the rule groups that have a large number of rules and match the network traffic only a few times should be separated from others."[3] This they say was through their observation of the fact that "that if rules with value v for a protocol field are grouped separately from others, then for any packet that does not have value v for the protocol field, we can quickly reject all those rules, and if only a few packets have that value, then those rules will be rejected most of the time."[4] With this idea they are able to maintain a small number of working-groups in memory for a particular workload.

2.1.1.2 Evaluation of WIND

The authors of this paper evaluated their work on a number of publicly-available datasets and on traffic from a border gateway router at a large academic institution. With these datasets they compared the real-time performance of WIND with existing IDS (Snort 2.1.3) using two metrics: "the number of packets processed per second and the amount of memory consumed."[5] Snort used in the evaluation contained about 2,059 rules and was run using the default configuration. The factor by which WIND improved the number of packets processed in
a second is shown in the figure 2.1 below:

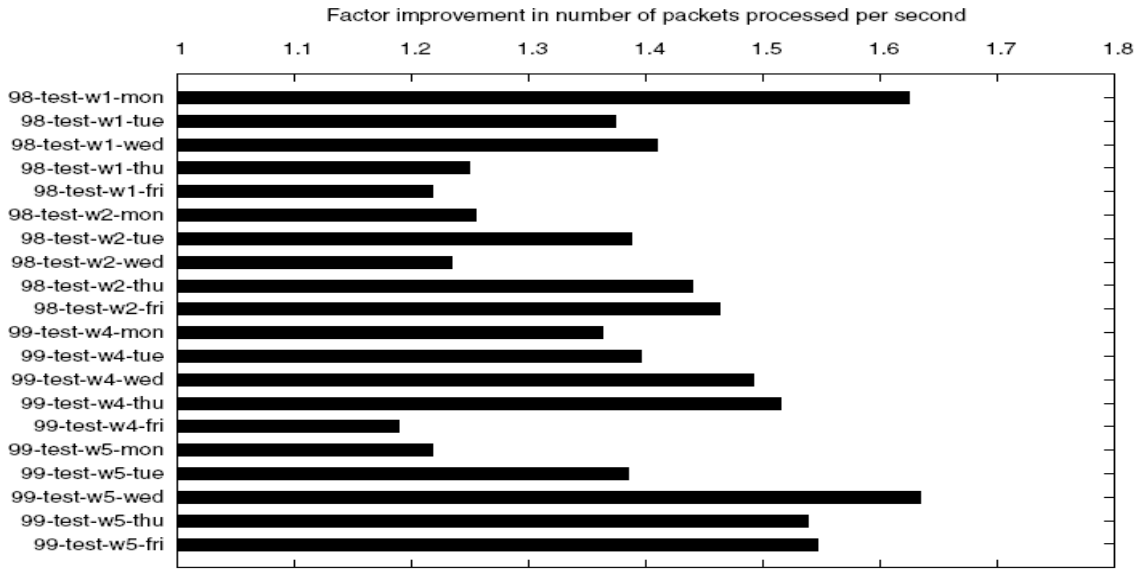


Figure 2.1 Factor Improvement, in terms of packets processed per second.

From the figure WIND was able to process up to 1.65 much faster than Snort 2.1.3. In terms of memory, they found out that WIND consumed 10 - 15% less memory than Snort as shown in the diagram below.

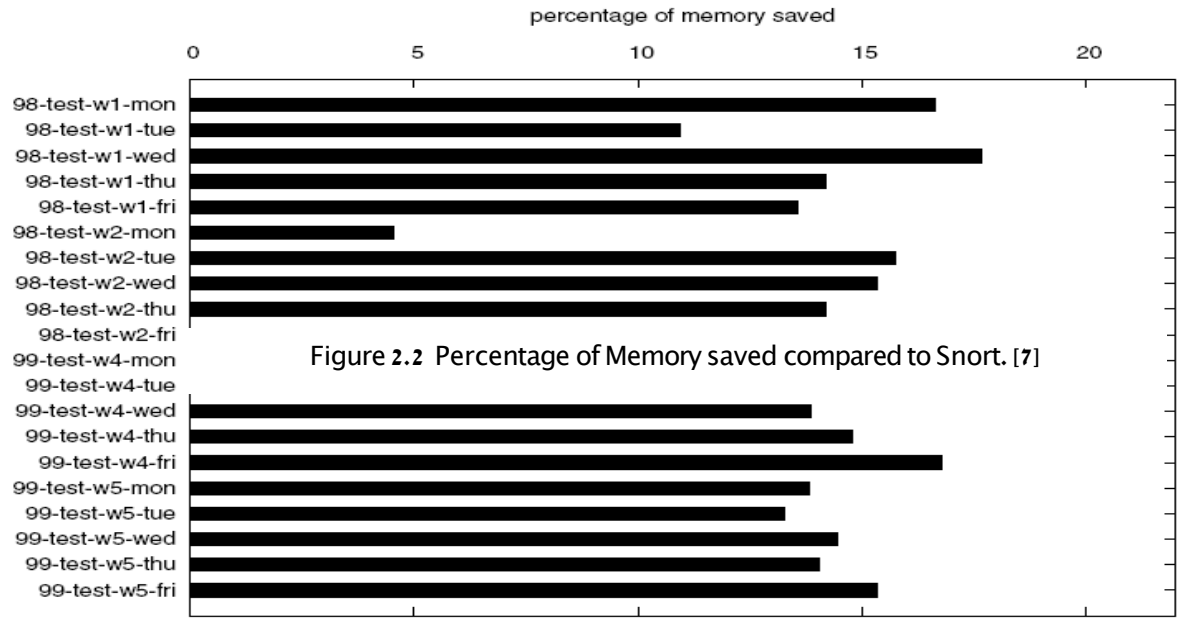


Figure 2.2 Percentage of Memory saved compared to Snort. [7]

2.1.2 Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading by *Bart Haagdorens, Tim Vermeiren and Marnix Goossens* [8]

Most of the existing Network Intrusion Detection System (NIDS) are implemented to match incoming packets against some known set of signatures using only a single thread of execution and as a result of this, NIDS benefit very little from multi-processor hardware platforms. It is in the light of this that the authors of the paper have proposed a way to improve the performance of NIDS by using multithreading through a series of designs. This paper first introduces the single-threaded model used in signature-based

NIDS sensors and from this model, they are able to come out with five different multithreaded design. The single-threaded model design is shown Figure 2.3 below:

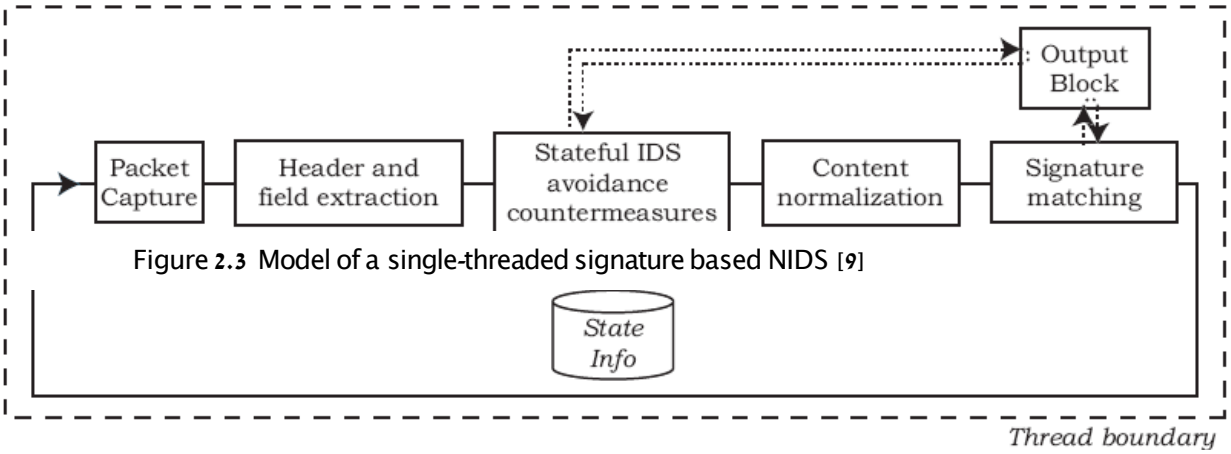


Figure 2.3 Model of a single-threaded signature based NIDS [9]

The model consists of a packet capture state which takes in the input network traffic and makes it available for processing; header and field extraction which examines the network packet header to identify the various fields for the various protocols; stateful IDS avoidance countermeasures which keeps the state information for different packets of the same connection; content normalization which is responsible for converting the content of an incoming packet to a standard representation for faster processing and lastly the signature matching state is where all incoming packets are matched to all the rules used by the IDS and if any match is found an alert is sent to the output block. Based on this design, the authors of this paper have suggested five different designs using multithreading. These designs are shown below.

2.1.2.1 Design 1: Separate Output Block

In this design the output block is separated to run on a different thread and a FIFO event queue buffer is used. By doing this the design is now able to isolate the output

latencies (caused by generated alerts) into a separate thread. The design is shown in Figure 2.4 below.

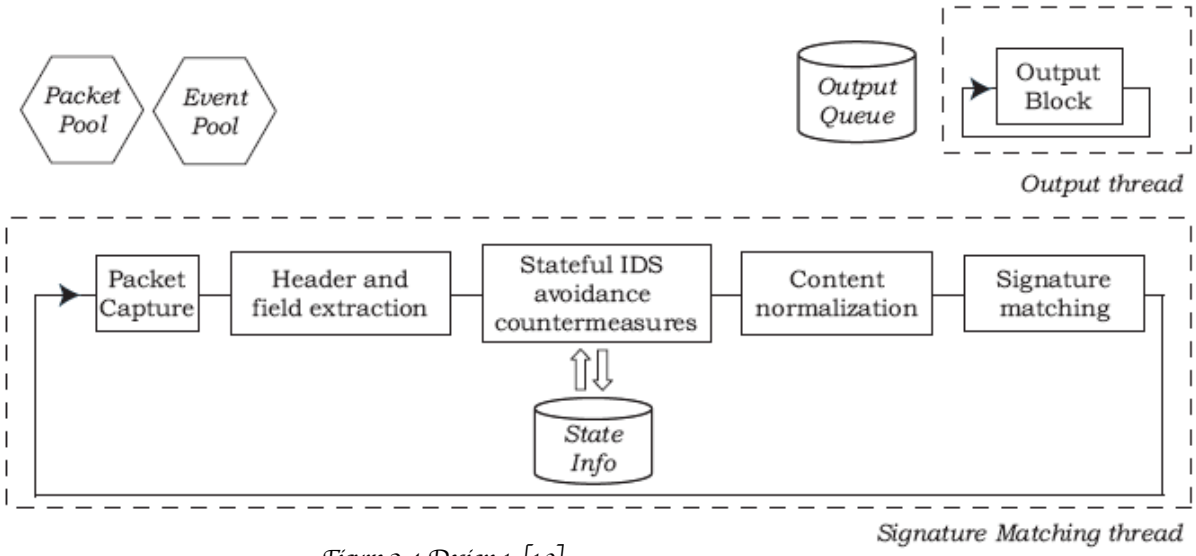


Figure 2.4 Design 1 [10]

2.1.2.2 Design 2: Parallel Signature Matching

This design separates the single packet processing state and runs a parallel signature matching thread. After the packets exist the content normalization block, they go into a matching queue and are taken out by the signature matching thread. With this design the most time-consuming block can executed in parallel threads potentially running on different CPUs and further this design introduces less overhead as compared with design 1 since the matching block do not depend on the state information. The design is shown in Figure 2.5 below.

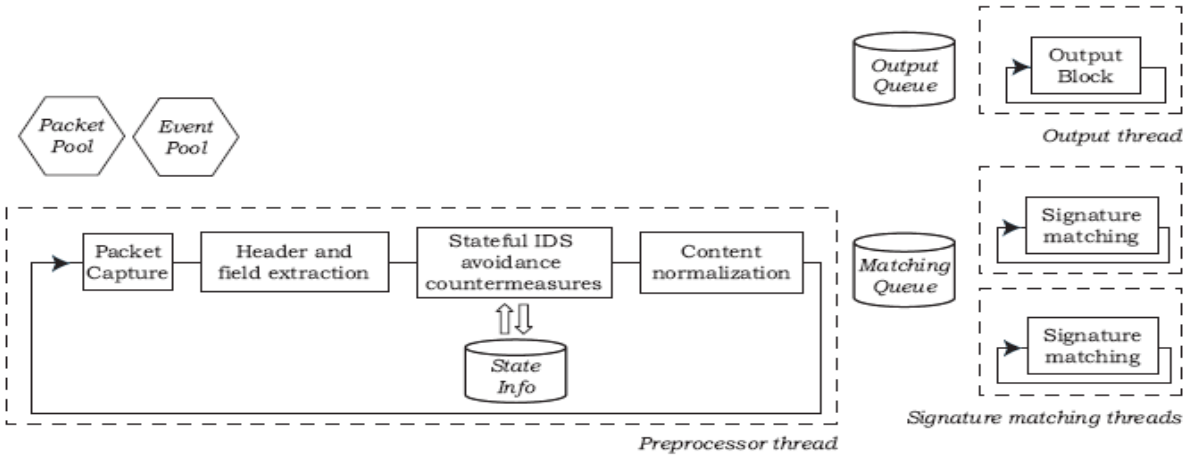


Figure 2.5 Design 2 [11]

2.1.2.3 Design 3: Parallel Content Normalization and Signature Matching

This design is similar to design **2** except that the content normalization blocks are moved into the signature matching threads to allow some extra code to be executed in parallel threads. The design diagram is shown in Figure **2.6** below.

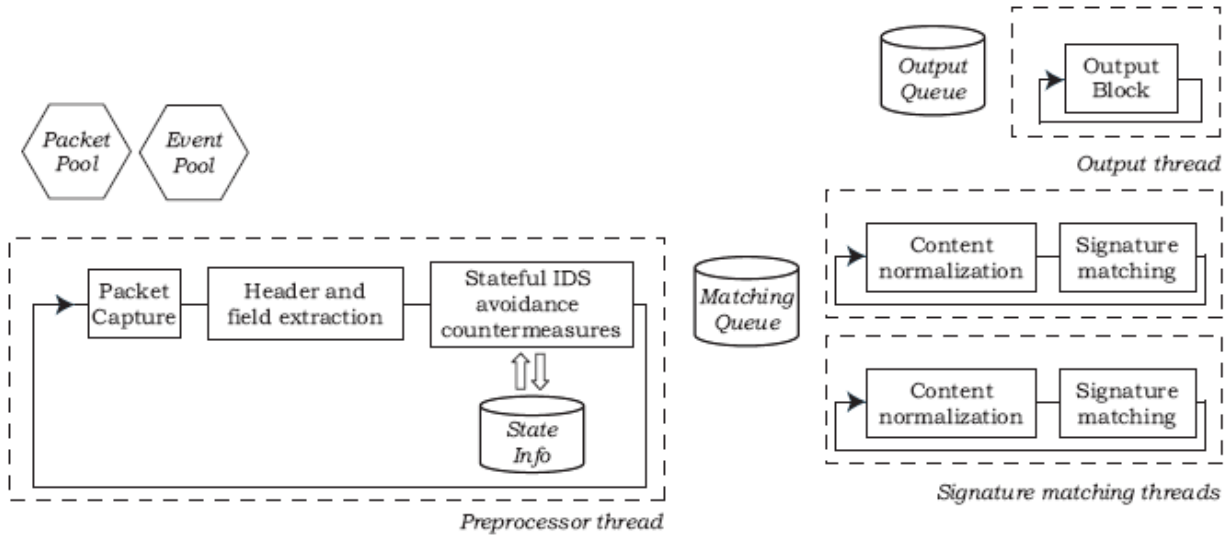


Figure 2.6 Design 3 [12]

2.1.2.4 Design 4: Parallel Stateful Countermeasures, Content Normalization and Signature Matching

This design further moves the stateful avoidance countermeasures into the signature matching loops. With this design two issues are introduced; firstly the same state information can be potentially accessed from different threads and secondly due to multiple threads being executed in parallel, it is possible that the stateful algorithm does not process the packet in the order they came in. The first issue was resolved by using synchronization and the second was solved by introducing the "order-of-Seniority Processing" (OoSP) conflict resolution block. The design is shown in Figure 2.7 below.

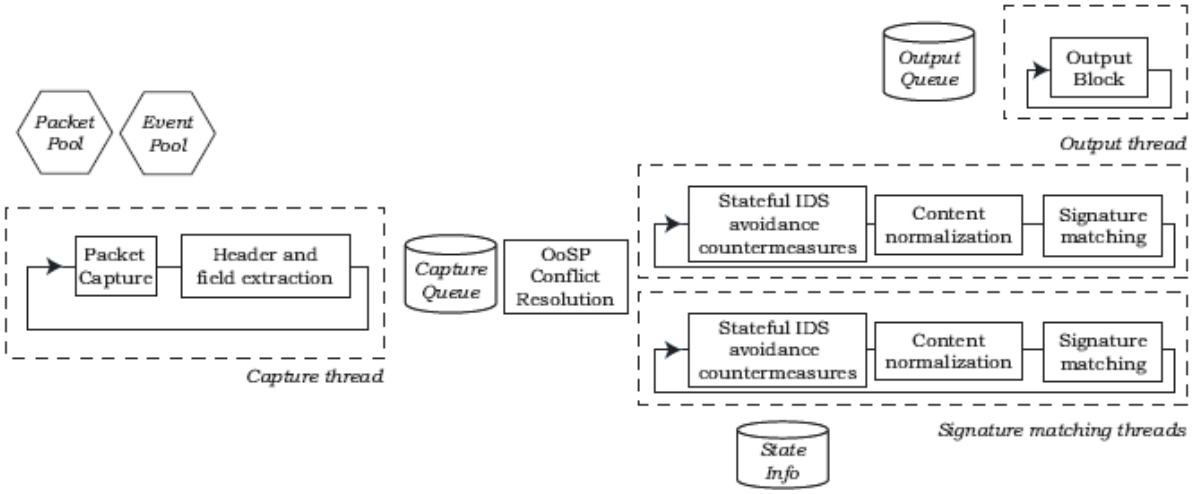


Figure 2.7 Design 4 [13]

2.1.2.5 Design 5: Parallel stateful Countermeasures with Separate Parallel Content Normalization and Signature Matching

This design moves the stateful IDS avoidance countermeasures and content normalization into a different thread which limits the impact of the OoSP requirement created by the countermeasures block. In this design after the packets go through the countermeasures block, they are stored in the matching queue and they can be processed without the requirement imposed by the OoSP. The design is shown in Figure 2.8 below.

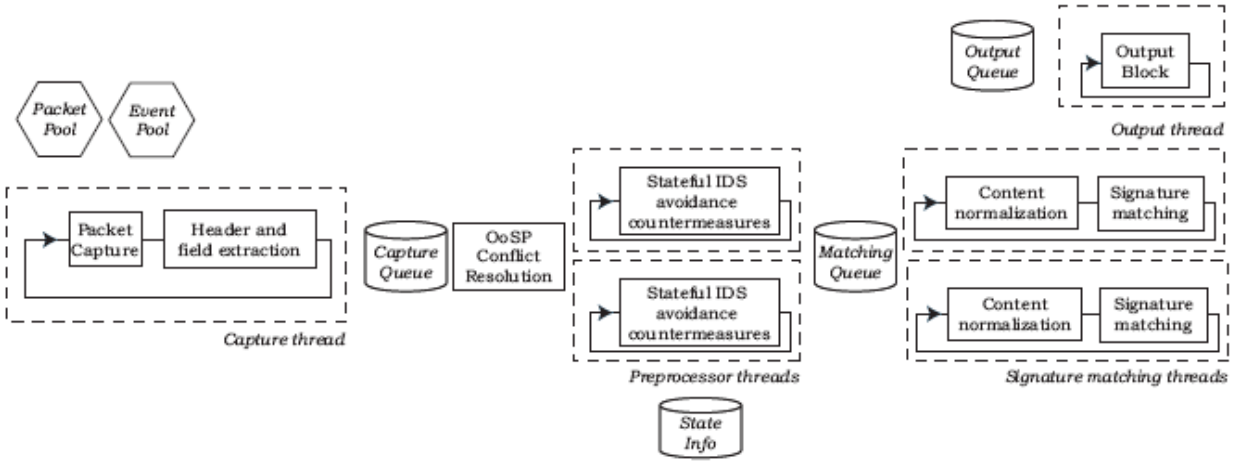


Figure 2.8 Design 5 [14]

2.1.2.6 Evaluation

The five designs were implemented through the modification of Snort **2.0.0** release and the multi-threading primitives were implemented using Posix threads (pthreads) API. The implementations were all evaluated using the following:

Dual Intel Xeon @ **2.84**GHz, **400**MHz bus, **512**KB L**2** cache and **1**GB of RAM running RedHat Linux **9** with a RedHat **2.4.20** and glibc **2.3.2** and all the experiments were carried out with the **1458** default rules of Snort's **2.0.5** release. Their results are shown below.

Run time (s)	single Xeon, Hyper-Threading disabled	dual Xeon, Hyper-Threading disabled	dual Xeon, Hyper-Threading enabled
Snort 2.0.0	132.1 (100%)	131.8 (99.8%)	131.6 (99.6%)
Design 1	163.6 (123.8%)	162.7 (123.2%)	162.4 (122.9%)
Design 2	189.8 (143.7%)	124.3 (94.1%)	113.7 (86.1%)
Design 3	180.2 (136.4%)	117.6 (89.0%)	111.0 (84.0%)
Design 4	222.4 (168.4%)	206.4 (156.2%)	199.9 (151.3%)
Design 5	223.9 (169.5%)	173.6 (131.4%)	159.0 (120.4%)

Table 1 Comparison of the cumulated run time percentage [15]

As table 1 shows design 2 and 3 achieved the best results in the multithreaded machine which out performs the single-threaded implementation by 16%. Design 4 and 5 carry a lot of overheads because of the complex design, more threads and shared data and hence could not outperform the single-threaded implementation. Furthermore design 5 outperforms 4 due to the impact of the OoSP requirement in design 4, which is more than in the design 5. Figure 2.9 below shows the run times for the various designs running on the dual Xenon with Hyper-Threading enabled.

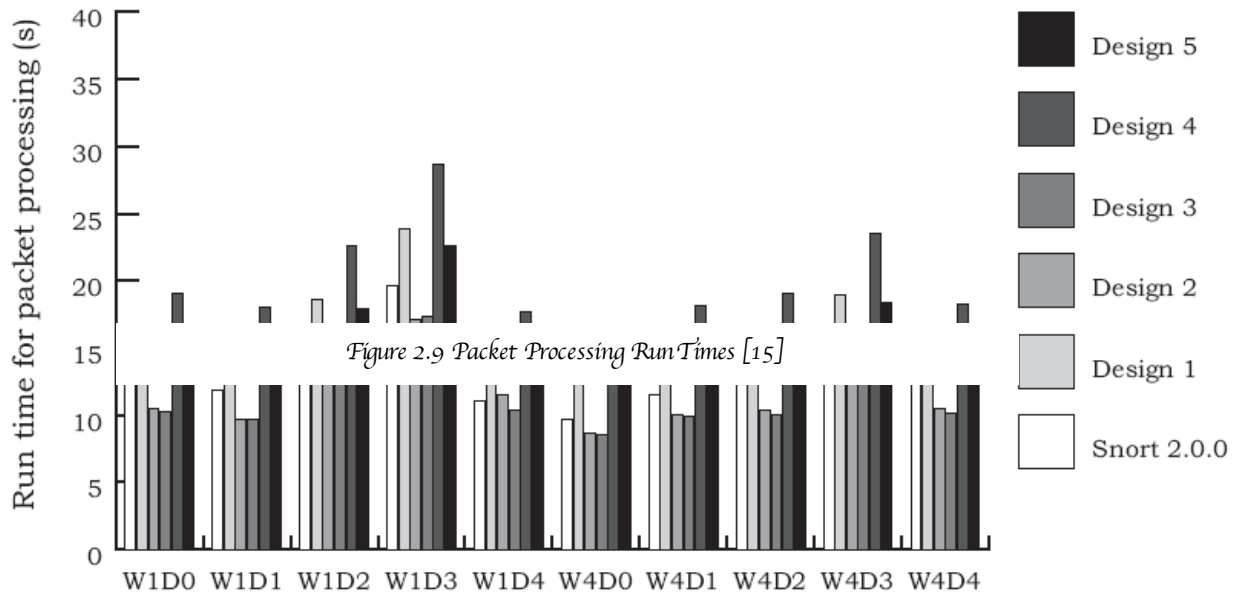


Figure 2.9 Packet Processing Run Times [15]

2.1.3 Snort Offloader: A Reconfigurable Hardware NIDS Filter by *Haoyu Song, Todd Sproull, Mike Attig, John Lockwood.* [16]

The authors of this paper argue that most of the Network Intrusion Detection Systems (NIDS) out there are implemented in software and often these software systems fail to keep up with the increasing high-speed of today's networks. Further they argue that most of the time used by these NIDS are spent doing rule or signature matching. For example, they stated that Snort 2.3.2 which contains over 2,600 rules consumes more than 80% of the CPU time just for string matching task alone and therefore as the traffic on a network increases software based solutions cannot process all the traffic in real-time. Hence they propose an "FPGA-based pre-filter" that can reduce the amount of traffic that will be sent to a software-based NIDS for matching. The key to their design is based on the observation that "malicious packets typically count only a small portion of the background "normal" traffic, yet they need enormous effort to figure out."[17] Therefore if a way can be found to isolate these "suspicious" packets then only these packets need to be further examined by the NIDS and the majority of the "normal" packets can be allowed to go without being examined and hence greatly reducing the burden on the NIDS. The architecture of their design is shown below.

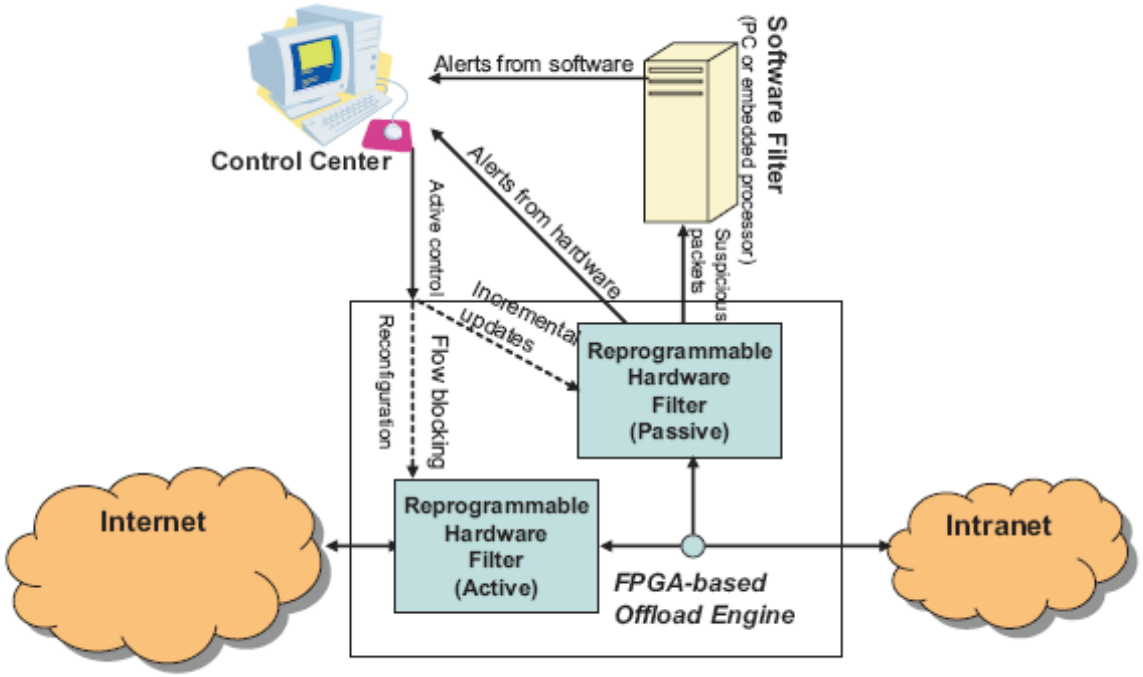


Figure 2.10 System Architecture [18]

The architecture consists of an FPGA-based Offload Engine which sits in-between the internal network and the external network to monitor all the traffic through the system; a control center which dynamically reconfigures the hardware through the network to update the filter set; an active filter which is used to block some flows based on the header of the packet and a passive filter which monitors both the header and payload of the packet and then passes suspicious packets to the software. The software is responsible for generating alert messages once a rule is matched to the control center while the hardware is responsible for header-only rule matching and sends alerts directly to the control center once a match is made.

2.1.3.1 Evaluation

This architecture was evaluated using modified rule sets of Snort of fixed-length content inspection, no regular expressions and no multiple content strings per rule. The experiment was conducted using real traffic from the Washington University campus. The diagram below shows amount of potential bandwidth saving on the amount of data sent to Snort running on a PC. From the diagram, it can be seen that on an average, traffic is reduced by 87%.

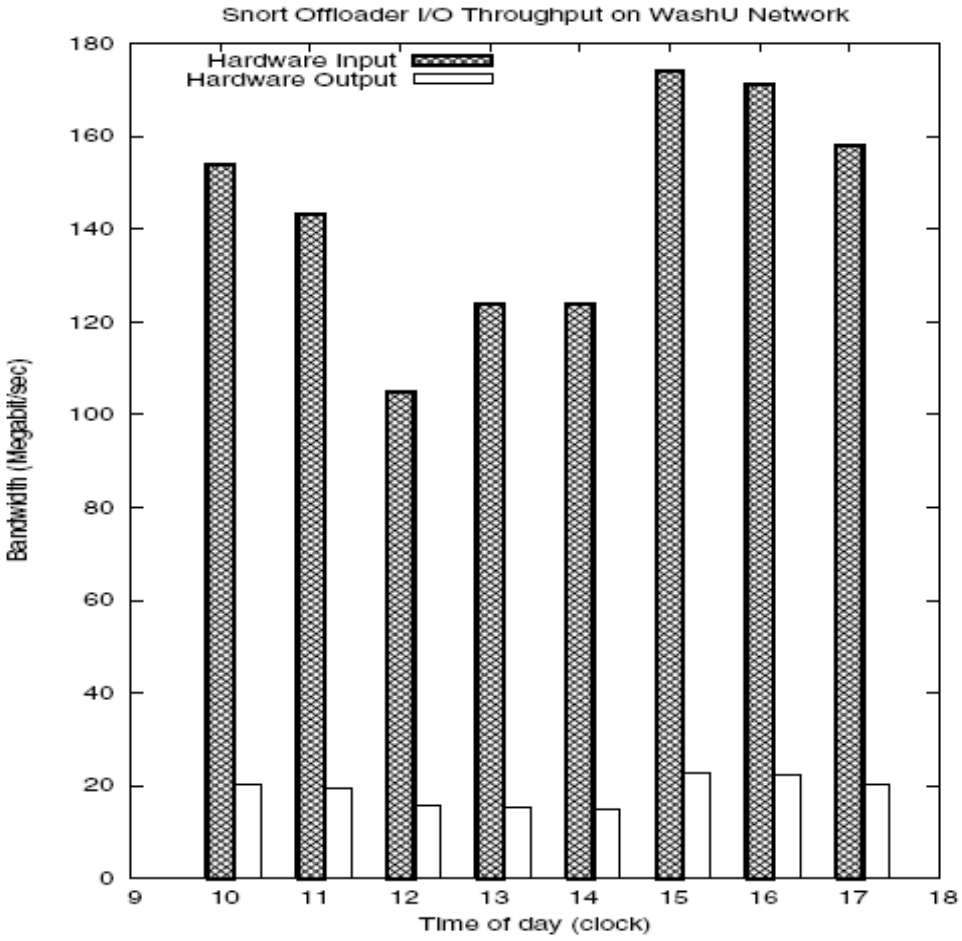


Figure 2.11. Network Traffic Bandwidth to and from Hardware Offload Engine. [19]

2.2.0 What is Snort?

Snort is a packet sniffer, a packet logger or a network intrusion detection system. It is an Open Source Software (OSS) which means that the source code is available to anyone to freely modify and because of this, Snort has become one of the most popular IDSs deployed in today's networks. The Snort architecture has been divided into four basic components which are:

- The sniffer
- The preprocessor
- The detection engine
- The output

The preprocessor, the detection engine and the alert components of Snort are all plug-ins meaning that they are not part of the Snort source code but separate programs which are written to conform to the Snort's plug-in API. In simple terms, Snort operates by firstly using its sniffing ability to capture packets from a network backbone and then processing the packets through the preprocessor which are then checked against a series of rules in the detection engine and if there are any matches found an alert message is sent to the configured output system. This process is shown in Figure **2.12** below.

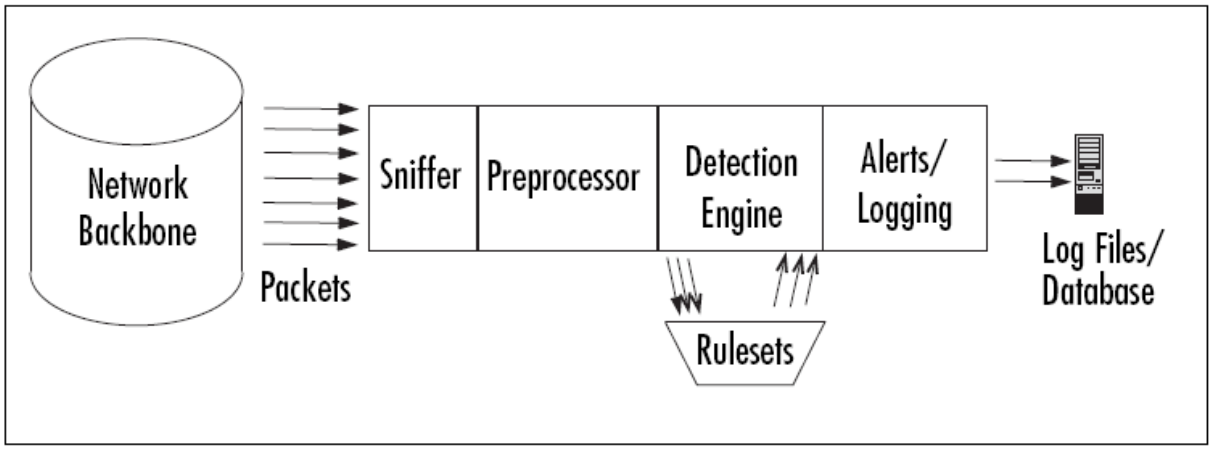


Figure 2.12 Snort Architecture [20]

2.2.1 Snort as a Packet Sniffer

A packet sniffer is device which is implemented either software or hardware that can be used to observe network traffic. In this mode Snort is able to save the packets that its captures in a configured logger. The packet-sniffing ability of Snort is shown in Figure **2.13**

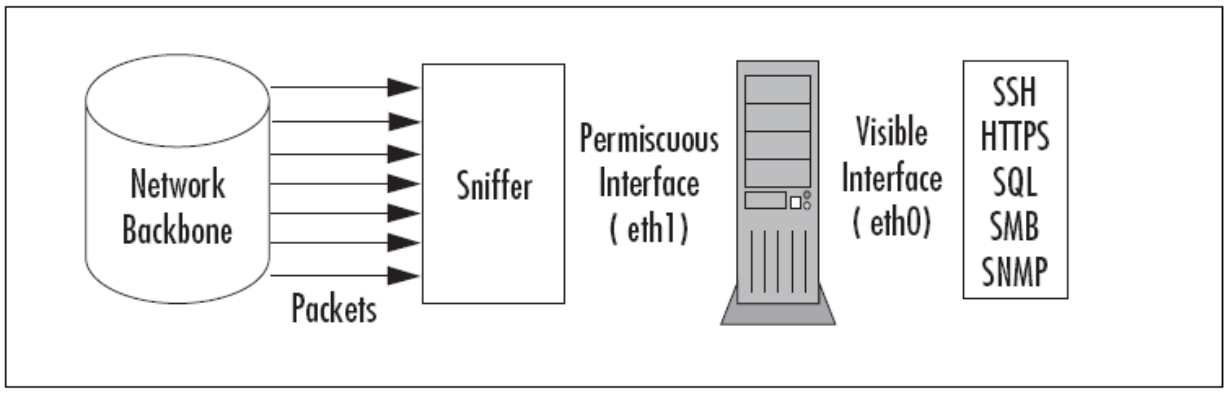


Figure 2.13 Snort's Sniffing Functionality[21]

2.2.2 The Preprocessor

The preprocessor is responsible for taking the raw packets from the sniffer and checking them against certain plug-ins such as an HTTP plug-in or a port scanner plug-in to check for some common "behavior" of these packets. Once the packet has been determined to have these types of behaviors, they are then forwarded to the detection engine to be examined further against the rule set. Figure 2.14 illustrates the workings of the preprocessor within Snort.

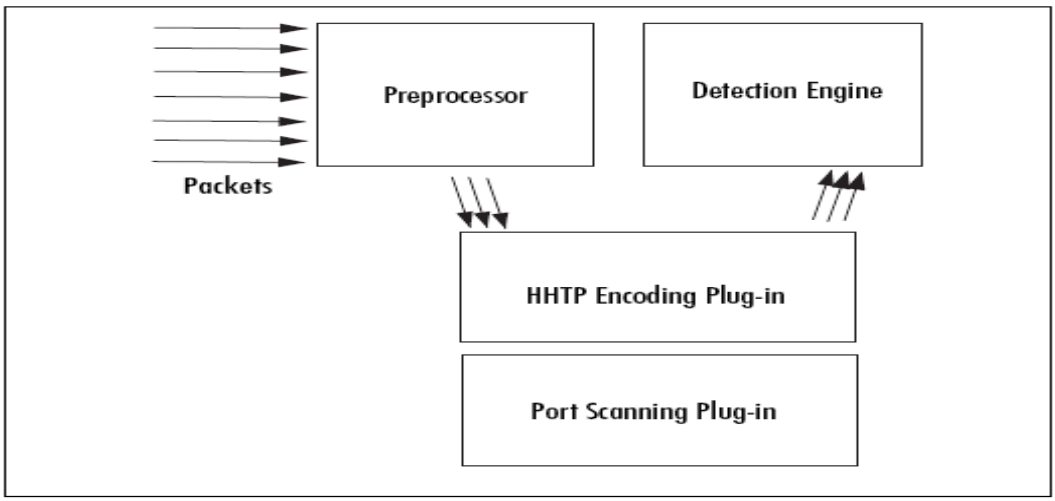


Figure 2.14 Snort's Preprocessor [22]

2.2.3 The Detection Engine

After the packet exits the preprocessors, they enter the detection engine which forms the core of the signature-based IDS in Snort. The detection engine is responsible for taking the packets that came from the preprocessor and then checking them against a

number of rule sets and if any of the rules matches the data in the packets, they are then passed to the alert processor to generate alert messages. The detection engine is illustrated in Figure **2.15** below.

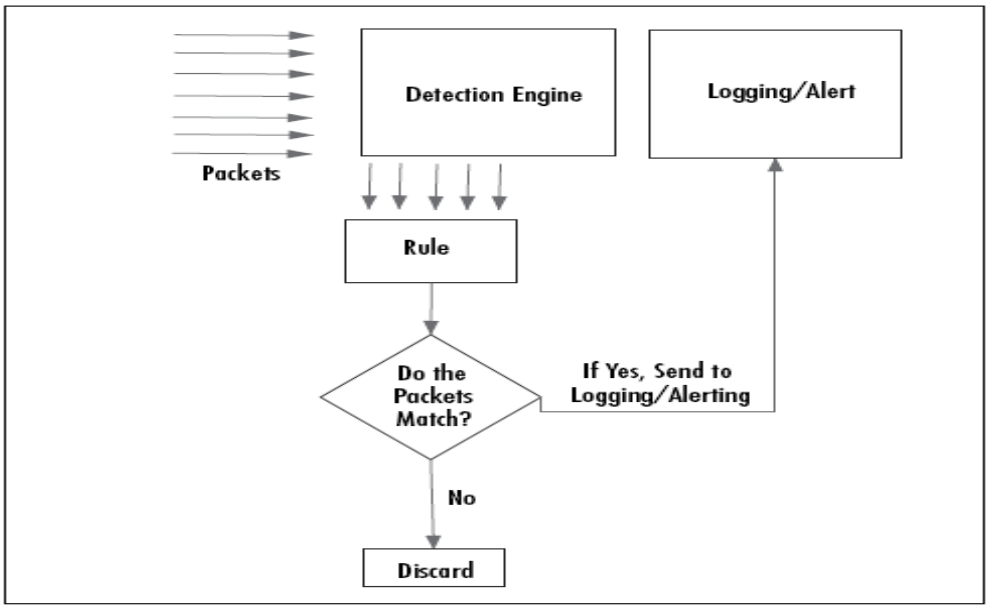


Figure 2.15 Snort's Detection Engine [23]

2.2.4 Snort's Output (Alerting/Logging)

After data has been matched against all the rules in the detection engine, if a match occurs, an alert will be triggered. Snort provides many ways to store this information; they can be sent to a log file, to the console or even to a configured database such MySQL and Postgres. The diagram below (Figure **2.16**) illustrates the many forms in which Snort can output its alert messages.

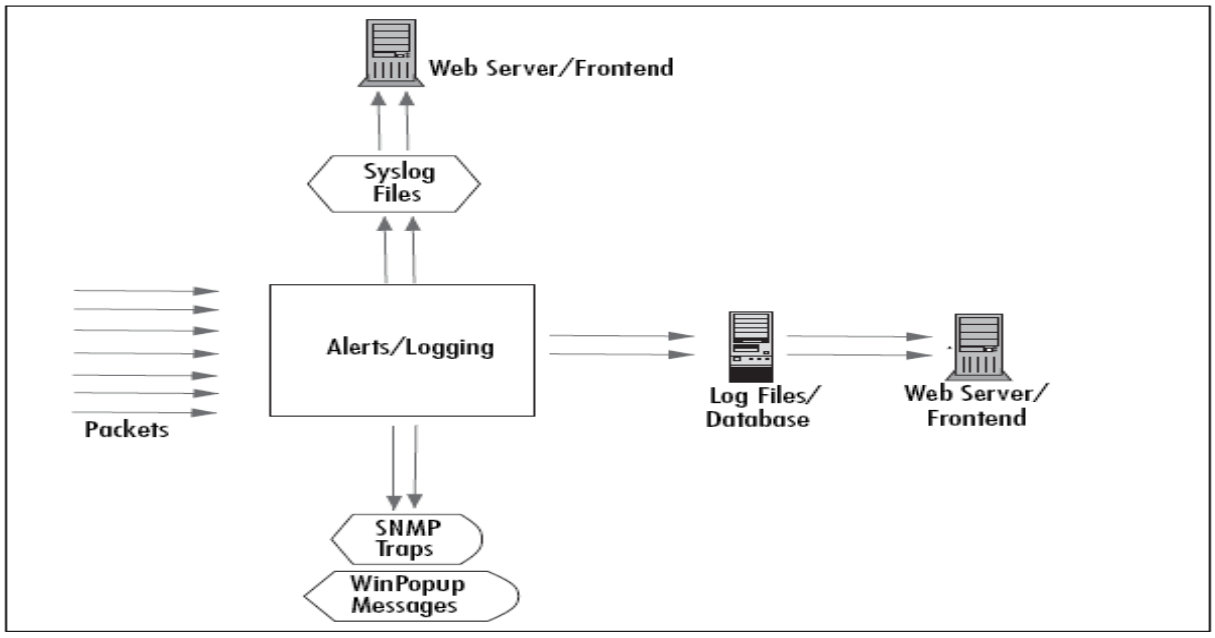


Figure 2.16 Snort's Alerting Component [24]

2.2.5 Snort Rules

Snort rules form the core part of the detection engine of Snort IDS. It is these rules that Snort uses to match against all incoming packets through a network infrastructure. Each packet is matched against all of the configured rules in the Snort configuration file and if there is a match, an appropriate action defined for that particular rule is taken which can be an alert message being generated or simply the packet being dropped. The Snort rules consist of two major parts:

- The rule header which defines the action to take when a match is found, the type of network protocol (e.g. TCP, UPD, ICMP etc.), the source and destination IP addresses and source and destination ports.
- The rule option which defines how the rule should match the content of the packet.

The basic form/syntax of a Snort rule is as follows:

Action Protocol Source IP Source Port -> Destination IP Destination Port Msg(options)

An example can be:

alert tcp 192.168.1.0/24 any -> 192.168.2.0/24 5000 (content: "|00 01 a5|"; msg: "mountd access"; sid:999; rev:2;)

2.2.5.1 The Rule Header

The rule header consists of all the parts with the exception of the "Msg" or the options (i.e. everything up to the parenthesis). The rule header is shown in bold in the following rule example.

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS CodeRed v2 root.exe access"; flow:to_server,established; uricontent:"/root.exe"; nocase; reference:url,www.cert.org/advisories/CA-2001-19.html; classtype:web-application-attack;sid:1256; rev:8;) [25]

- **Rule Actions:** The rule action defines what course of action is to be taken when the particular rule matches an incoming packet. Currently Snort has about eight defined actions but the most common ones are the *alert* and *pass*. The alert action generates events (base on how the output/alert plug-in is configured) when a match is found and the pass rule allows a packet go through without being further processed by Snort. If Snort is running in the inline mode, you can also define these actions: *drop*, *reject* and *sdrop* (silent drop).

- **Protocols:** The next field in the rule header is the protocol. This field can either be TCP, UDP, ICMP or even just IP to cover for all the three protocols. To look for other protocols other than IP, the proto option within a rule can be used.

- **Source and Destination IP Addresses:** These fields define the source and destination IP addresses respectively. Snort allows for using variables (by using the keyword "var") in defining these fields. The format is var <variable name> <value>. For example:

var HOME_NET 192.168.20.0/24

var EXTERNAL NET any (snort allows the use of the keyword "any" to refer to all network just like in an access list).

alert ICMP $HOME_NET any -> $EXTERNAL_NET any

- **Source and Destination Ports:** These fields define the source and destination ports respectively. These can be defined as a single port value or a range of port values. For example to match on any port from say 50 to 100 inclusive, you can specify 50:100. The not (!) operator can also be used to match on all ports except one. For example to match on any port except 23, you will specify !23. Snort also allows the use of the keyword any to refer to any port.

2.2.5.2 The Rule Options

The portion of the rule that occurs within the parenthesis forms the rule options. Although Snort will not complain about how these options are ordered, the order can be very important to the accuracy and performance of the rule. Options are separated from each other by the use of a semi-colon (";"). The rule options are shown in bold in the following example.

alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
(msg:"WEB-IIS CodeRed v2 root.exe access"; flow:to_server,established; uricontent:"/root.exe"; nocase; reference:url,www.cert.org/advisories/CA-2001-19.html; classtype:web-application-attack;sid:1256; rev:8;)

The following forms the rule options.

- **The Rule Title:** In the example above the first option is the rule title *msg* which means message or the rule title. This defines the text that will be sent to the output processor when a match is made on a packet. This field can be duplicated among many rules which means this field cannot uniquely identify a rule. The text message must be enclosed in quotes.

Flow: This particular option becomes very important in helping to control the load on Snort when it comes to matching the content of packet. The predefined flows are *to_server, from_server, to_client, established* and *stateless.* Some of these flows mean the same thing for example from_server and to_client. Using these one can reduce the content matching of Snort. For example using a flow of established will tell the detection to match only the packets that have been started by a full three-way TCP handshake. Any other stream of data will not be examined by any rule that has the flow set to "established".

- **Content and uricontent:** This is where most of the work done by Snort occurs. Both of these are very similar except that content matches in the payload of the packet while uricontent does the matching in the normalized output of the HTTP preprocessor. The content option can either be a plain text or as binary data in hex format using the pipes (|) inside quotation marks or combination of both. An example is shown given below.
  - content: "|00 45 99 E0|";
  - content: "|00 |some text|99 A2 00|";

  Snort also provides some other options which works with the content option. These are *depth* which tells where in the packet to look for a match (e.g. content: "come"; depth:12; this say it should match if the word "come" is in the first 12 bytes of the packet); *offset* which tells to skip that many bytes in the payload before doing the match (e.g. content: "root.exe"; offset:50; which says skip the first 50 bytes of the payload before doing the match). The offset can be combined with the depth option in which case the depth will start from the offset point. There are other options such within and distance which can also be used together.
- **Sid option:** This is a very important option since it is required for every rule for Snort to run; without this Snort will issue an error and exit. This is a number also know as Snort ID and must be unique for each rule that you define. Snort.org and VRT rule sets use Sid ranges 100-1,000,000 while ranges 1,000,001-1,999,999 are reserved for local use.

Rev option: This refers to the revision number; these are given once you update your rules. Combining this option with the Sid option will allow you to uniquely identify a rule. This option is also a number.

- **PCRE option:** PCRE stands for Perl Compatible Regular Expressions. Using this, one is able do very complex matches which cannot be done with the normal content matching. But this should be used with care because PCRE can be very CPU intensive and a poorly written PCRE can bring a CPU to its knees in no time.

2.2.6 Snort Configuration used in this Experiment

In this experiment Snort was used as a Network Intrusion Detection System (NIDS) and Snort version 2.6.1.5 (Build 59) was used and the rule sets used was from Snort version 2.5 together with custom rules generated during the course of this experiment using the simple java application. The configuration file (snort.conf) used in this experiment is the default that came with Snort 2.6.1.5 with just few changes such as setting the HOME_NET and EXTERNAL_NET variables and adding additional rules. There was no database configured for this experiment setup.

3.0 Methodology

3.1 Lab Setup

This experiment was conducted using three different machines. The experiment setup is shown in Figure 3.1 below.

The experiment was conducted in two different labs with the same setup but different end devices. Netload1 and netload3 were replaced with two sun blade machines. The end devices both had DITG running on them and Snort was installed on noseeum. All of these machines had gigabit network interfaces. The IP address assignments are shown in the table 2 below.

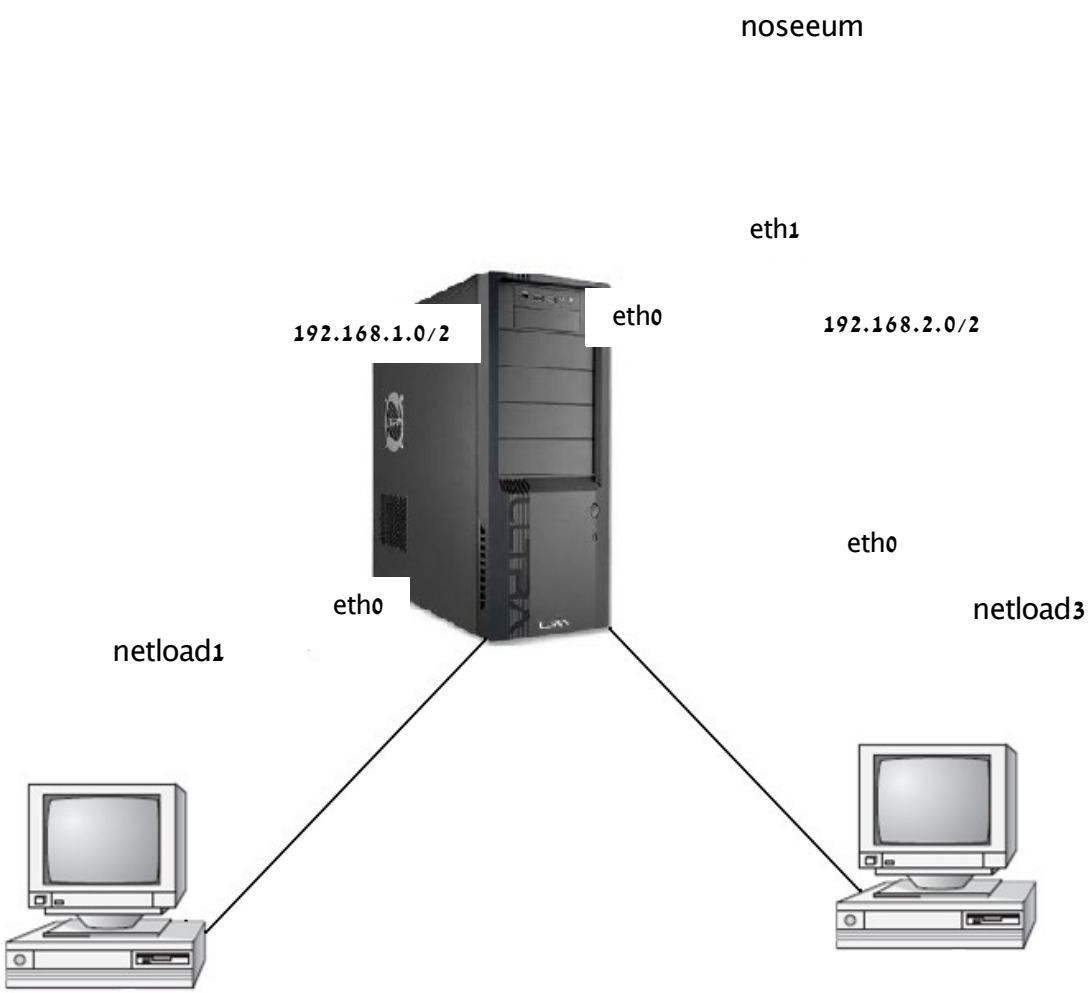


Figure 3.1 Lab Setup

Host Name/Interface	eth0	eth1
noseeum	192.168.1.1/24	192.168.2.1/24
Netload1	192.168.1.2/24	
Netload3	192.168.2.2/24	

Table 2. IPAddress Assignment

All of these machines were running Scientific Linux 2.6. Routing was done using static routes and enabling IP forwarding on noseeum. Traffic was generated using DITG from one end device (e.g. netload1) to the other end device (e.g. netload3) and this traffic went through noseeum which had Snort configured on it to examine all the packets being generated from one host to the other host. As stated earlier on, these experiments were conducted in two different labs with different end devices in terms of processing speeds. One was conducted in the Computing Science network lab with end device netload1 and netload3 with more processing power as compared with end devices (sun blades-rs3 and rs4) in the MINT lab which had more processing speed but in both cases noseeum was used.

There were two methodologies used in this experiment to get the desired results. The first was sending packets from one host to the other host through Snort running on noseeum and then watching for packet drops as well as measuring the packets delay. The number of packets dropped was computed by the difference obtained by observing the number of packets transmitted from the sender and comparing this number to the number of packets received at the receiver end. This method was inconclusive due to the fact that it was later discovered that Snort was actually letting most of the packets through without examining them and therefore when those two values are compared they were always equal to each other.

This lead to second approach where the statistics that Snort outputs when it exits was taken a look at. This output reveals the total number of packets received, the total number that Snort was able to examine, the total number of packets that were dropped (dropped here means passed without Snort examining it) and lastly the total number of packets outstanding. This second approach, resulted in being able to effectively investigate the desired results for this experiment by measuring the performance of Snort under different conditions such as increasing the load and increasing the number of rules. For testing whether Snort was allowing packets to pass through that it was suppose to drop, Snort was run in the inline mode with a rule to drop all packets of a

particular stream (say TCP). Wireshark was installed at both the sender and receiver end device to observe the packets at the receiver's end.

3.2 Rules used in the Experiment

The rules used in this experiment was downloaded from www.snort.org and Snort 2.5 rule sets which contains about 3,005 rules was freely available for download at the time of the experiment. In addition to these rules, a simple java application program was also developed that can randomly generate Snort rules using the Snort rule syntax. A screenshot of this program is shown in Figure 3.2 below.

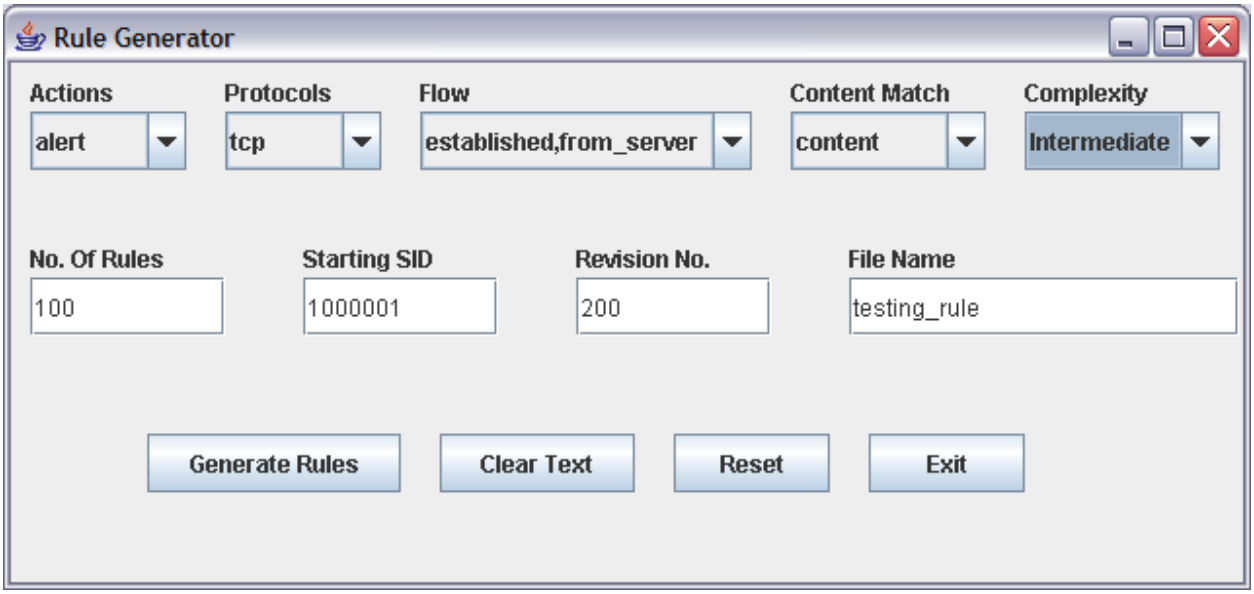


Figure 3.2 A Simple Java Application Snort Rule Generator

This application was able to randomly generate any number of rules. The rules generated were written to a local file with the specified name. The main aim of this application was to generate rules of varying complexities from simple to intermediate to complex. The simple form only play around with the rule header matching on the source and destination IP addresses, source and destination port numbers and the protocol field without going into the payload of the packet. The IP addresses were all randomly g

```
alert tcp 68.222.177any -> 189.5.241any (msg: "Just a simple test 0"; sid:1000001; rev:
200;)
alert tcp 108.213.175any -> 102.16.17any (msg: "Just a simple test 1"; sid:1000002;
rev: 200;)
alert tcp 3.63.244any -> 67.158.236any (msg: "Just a simple test 2"; sid:1000003; rev:
```

In the intermediate complexity the content option was added to the rule. A text file was also passed to this program to randomly pick up words to search in the content as well as randomly generating binary hex data. A sample output is shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 0"; content:"|F7 E8 C6 A2 D5 D2 A8 |"; depth:100; sid:1000001; rev:100;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 1"; content:"requires"; depth: 263; sid:1000002; rev:100;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 2"; content:"|E2 58 A6 25|z|92C4 33|x|97E2 48|o|39E3|"; depth:145; sid:1000003; rev:100;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 3"; content:"also"; depth:296; sid:1000004; rev:100;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 4"; content:"local"; depth:127;
```

The complex level is similar to the intermediate level except that more options are introduced and there can be more than one content option to match the payload against. A sample output is shown below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 0"; flow: stateless; dsize:>688; content:"|E9 62|n|82A8 00|s|14F8 05|z|59A9 87|b|63|"; offset:413; depth:143; nocase; content:"built"; within:19; distance:5; nocase; content:"true"; offset:335; depth:132; nocase; sid:1; rev:5;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 1"; flow: stateless; dsize:>656; content:"|B3 97|p|81F5 91|g|44C0 18|u|26F1 24|x|35F5 86|g|01F7 49|b|78A6|"; offset:315; depth:256; nocase; content:"|B5 D6 E5 A0 E8 F6 C3 E1 A9|"; within:46; distance:28; nocase; content:"given"; offset:330; depth:187; nocase; sid:2; rev:5;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg: "Just testing rules 2"; flow: stateless; dsize:>541; content:"systematically"; offset:295; depth:155; nocase; content:"|E8 D3 A5 C2 A7|"; within:12; distance:16; nocase; content:"test"; offset:492; depth:173; nocase; sid:3; rev:5;)
```

3.3 DITG Flows used in this Experiment

In all the experiments conducted, a total of 5 flows were used each with a constant rate of packets generation. A sample of a flow is shown below.

```
-a 192.168.2.2 -rp 1001 -C 1000 -c 512 -T TCP -t 10000
-a 192.168.2.2 -rp 1002 -C 1000 -c 512 -T TCP -t 10000
-a 192.168.2.2 -rp 1003 -C 1000 -c 512 -T TCP -t 10000
-a 192.168.2.2 -rp 1004 -C 1000 -c 512 -T TCP -t 10000
-a 192.168.2.2 -rp 1005 -C 1000 -c 512 -T TCP -t 10000
```

The script was used for all the experiments but with changing the rate of packets generated per second.

3.4 Experiment Devices Capabilities

The following are the capabilities (CPU and Memory) of all the devices used in this experiment.

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 6
model name      : AMD Athlon(tm) XP 1800+
stepping        : 2
cpu MHz         : 1533.107
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 mtrr pge mca cmov pat
pse36 mmx fxsr sse syscall mp mmxext 3dnowext 3dnow up ts
bogomips        : 3067.75

Memory: 1295MB

OS Version:Linux 2.6.18-8.1.14.el5
```

3.4.2 netload1

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 10
model name      : AMD Athlon(TM) XP 3000+
stepping        : 0
cpu MHz         : 2166.528
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov
pat pse36 mmx fxsr sse syscall mmxext 3dnowext 3dnow up ts
bogomips        : 4335.12

memory:1035MB

OS Version:Linux 2.6.18-8.1.6.el5
```

3.4.3 netload3

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 8
model name      : Pentium III (Coppermine)
stepping        : 3
cpu MHz         : 864.504
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 mtrr pge mca cmov pat
pse36 mmx fxsr sse up
bogomips        : 1730.05

memory:1035MB

OS Version:Linux 2.6.18-8.1.6.el5
```

3.4.4 rs3

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 15
model           : 39
model name      : AMD Opteron(tm) Processor 148
stepping        : 1
cpu MHz         : 1005.158
cache size      : 1024 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 syscall nx mmxext fxsr_opt lm 3dnowext 3dnow
pni lahf_lm
bogomips        : 2013.27
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management: ts fid vid ttp

Memory: 1027MB

OS Version: Linux 2.6.15-26-amd64-generic
```

3.4.5 rs4

```
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 15
model           : 39
model name      : AMD Opteron(tm) Processor 148
stepping        : 1
cpu MHz         : 1005.168
cache size      : 1024 KB
fpu             : yes
fpu_exception   : yes
cpuid level     : 1
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 syscall nx mmxext fxsr_opt lm 3dnowext 3dnow
pni lahf_lm
bogomips        : 2013.27
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management: ts fid vid ttp

Memory: 1027MB

OS version: Linux 2.6.15-26-amd64-generic
```

The first method was to send data from one end (netload1) using DITG sender and then using DITG receiver to receive those packets at the other end (netload3). The packets dropped were computed by taking the total packets received at the receiver end and subtracting it from the total packets transmitted at the sender using the ifconfig command. The table below shows the results from Computing Science Network Lab. This experiment was run using 3005 rules

Packet Rate	Packets Tx	Packets RX	Packets Dropped	Average Delay
80000	358811	358811	0	0.080043
90000	371266	371266	0	0.082528
100000	361322	361322	0	0.080590
110000	369547	369547	0	0.082850

Table 3 Statistics collected at the CS Lab

It was discovered that Snort was actually letting the packets pass through without examining them if there was a packet overload. This was shown in the output summary Snort prints to the screen when it exits with the Ctrl-C. Due to this, the way of gathering data was then modified. Instead of looking at the packets transmitted and received at the end points, the data that Snort outputs to the screen when it exits was collected. All the experiments were repeated using this method in the two labs. These set of tables represent the results from the CS lab with light load traffic and moderate number of rules.

Table 4a: Constant Packet Rate of 100/sec.				
Number of	Total	Packets	Packets	Packets

Rules	packets	Analyzed (%)	Dropped (%)	Outstanding (%)
101	30509	49.99508342	0	50.00491658
201	30424	49.99671312	0	50.00328688
613	30491	49.99508052	0	50.00491948
1022	30475	49.99507793	0	50.00492207
2205	30485	49.99507955	0	50.00492045

Table 4b: Constant Packet Rate of 500/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	150164	49.99866812	0	50.00133188
201	150213	49.99900142	0	50.00099858
613	150195	49.9990013	0	50.0009987
1022	150164	49.99933406	0	50.00066594
2205	150166	49.99933407	0	50.00066593

Table 4c: Constant Packet Rate of 1000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	116216	35.89350864	28.21040132	35.89609004
201	130004	37.97344697	24.05079844	37.97575459
613	130644	38.38829185	23.22111999	38.39058816
1022	132477	37.11361217	25.77126596	37.11512187
2205	129442	39.32185844	21.35396548	39.32417608

Table 4d: Constant Packet Rate of 5000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	340819	13.91383696	72.17144584	13.9147172
201	342459	13.19340417	73.61231563	13.19428019
613	353066	11.93148023	76.13618983	11.93232993
1022	337317	13.11021976	73.7786711	13.11110913
2205	348281	13.17011264	73.65920047	13.17068689

Table 4e: Constant Packet Rate of 10000/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	343265	12.23602756	75.52707092	12.23690152
201	349140	11.41146818	77.17649081	11.41204102
613	360779	11.60766009	76.78412546	11.60821445
1022	354748	12.63375692	74.73164049	12.63460259
2205	350955	11.78783605	76.42347309	11.78869086

The following tables show the results gotten from running the experiment in the MINT lab under the same conditions.

Table 5a: Constant Packet Rate of 100/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	8161	49.9816199	0	50.0183801
201	8145	49.98158379	0	50.01841621
413	8152	49.98773307	0	50.01226693
613	8152	49.98773307	0	50.01226693
814	8167	49.9816334	0	50.0183666
1022	8162	49.9877481	0	50.0122519
2205	8153	49.98160186	0	50.01839814

Table 5b: Constant Packet Rate of 500/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)

101	38238	49.9973848	0	50.0026152
201	39345	49.93773033	0	50.06226967
413	39204	49.99744924	0	50.00255076
613	34936	49.93130295	0	50.06869705
814	39323	49.99618544	0	50.00381456
1022	35610	49.9971918	0	50.0028082
2205	37950	49.99736495	0	50.00263505

Table 5c: Constant Packet Rate of 1000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	56927	38.20331301	23.5898607	38.20682629
201	61514	40.24287154	16.25971324	41.87176903
413	60105	40.36935363	19.17810498	40.45254139
613	59948	40.92880496	18.13738573	40.9338093
814	59429	40.02423059	19.87077016	40.10499924
1022	60044	40.49863433	18.99940044	40.50196523
2205	59739	40.49448434	19.00768342	40.49783224

Table 5d: Constant Packet Rate of 5000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	186528	33.04490479	33.90858209	33.04651312
201	248523	40.47110328	19.05658631	40.47231041
413	219464	36.90445813	26.18971676	36.9058251
613	211421	35.37586143	29.24733115	35.37680741
814	208310	33.14435217	33.68681292	33.16883491
1022	222210	36.67791729	26.64326538	36.67881733
2205	226688	38.49784726	22.98224873	38.51990401

Table 5e: Constant Packet Rate of 10000/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
101	315283	28.33390954	43.31663934	28.34945113
201	337078	21.51638493	56.96634013	21.51727493
413	344031	21.02717488	57.9450689	21.02775622
613	324101	26.91722642	46.16493007	26.91784351
814	326123	19.17436059	61.63625381	19.1893856
1022	334555	26.08748935	47.82442349	26.08808716
2205	309021	14.29417418	71.39579511	14.31003071

The following tables show the results at high loads in the CS lab.

Table 6a: Constant Packet Rate of 20000/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	349916	10.83202826	78.33508613	10.83288561
2205	351029	12.13062169	75.73818687	12.13119144
3205	347820	10.56408487	78.87125525	10.56465988
4205	339196	9.780775717	80.43756412	9.781660161

Table 6b: Constant Packet Rate of 40000/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	351213	11.02037795	77.95838992	11.02123213
2205	342380	10.47870787	79.04200012	10.47929201
3205	347281	9.355824246	81.28748765	9.356400149
4205	342177	10.9712225	78.05697052	10.97180699

Table 6c: Constant Packet Rate of 60000/sec.				
Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	351898	11.70538054	76.58838641	11.70623306
2205	354856	13.94340239	72.1126316	13.943966
3205	339160	9.071824508	81.85546645	9.072709046

4205	335731	10.13102752	79.73705139	10.13192109

Table 6d: Constant Packet Rate of 80000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	341008	11.25340168	77.4923169	11.25428142
2205	323397	10.71809572	78.56288092	10.71902337
3205	343357	11.6185195	76.76208727	11.61939323
4205	341574	10.97858736	78.04165422	10.97975841

Table 6e: Constant Packet Rate of 100000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	348901	11.39922213	77.2006959	11.40008197
2205	347337	12.60562508	74.78788612	12.6064888
3205	348937	11.37655221	77.24603582	11.37741197
4205	339117	9.831415116	80.33628512	9.832299767

The following the tables show the results in the MINT lab under same conditions.

Table 7a: Constant Packet Rate of 20000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	394679	16.95985852	66.07977622	16.96036526
2205	522092	0.250530558	99.49855581	0.250913632
3205	521821	0.249510848	99.5004034	0.250085757
4205	523210	0.138185432	99.72305575	0.138758816

Table 7b: Constant Packet Rate of 40000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)

1022	542527	0.422098808	99.15543374	0.422467453
2205	643701	0.254932026	99.48982524	0.255242729
3205	647127	0.140930606	99.7176752	0.141394193
4205	639111	0.117350507	99.76498605	0.117663442

Table 7c: Constant Packet Rate of 60000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	623693	0.633965749	98.73174783	0.63428642
2205	634954	0.200329473	99.59556125	0.204109274
3205	640345	0.201922401	99.59584287	0.202234733
4205	639881	0.20081859	99.59805026	0.201131148

Table 7d: Constant Packet Rate of 80000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	631353	0.663178919	98.67316699	0.663654089
2205	637024	0.226051138	99.54020571	0.233743156
3205	640853	0.189591061	99.62050579	0.189903145
4205	640127	0.18590061	99.62773012	0.186369267

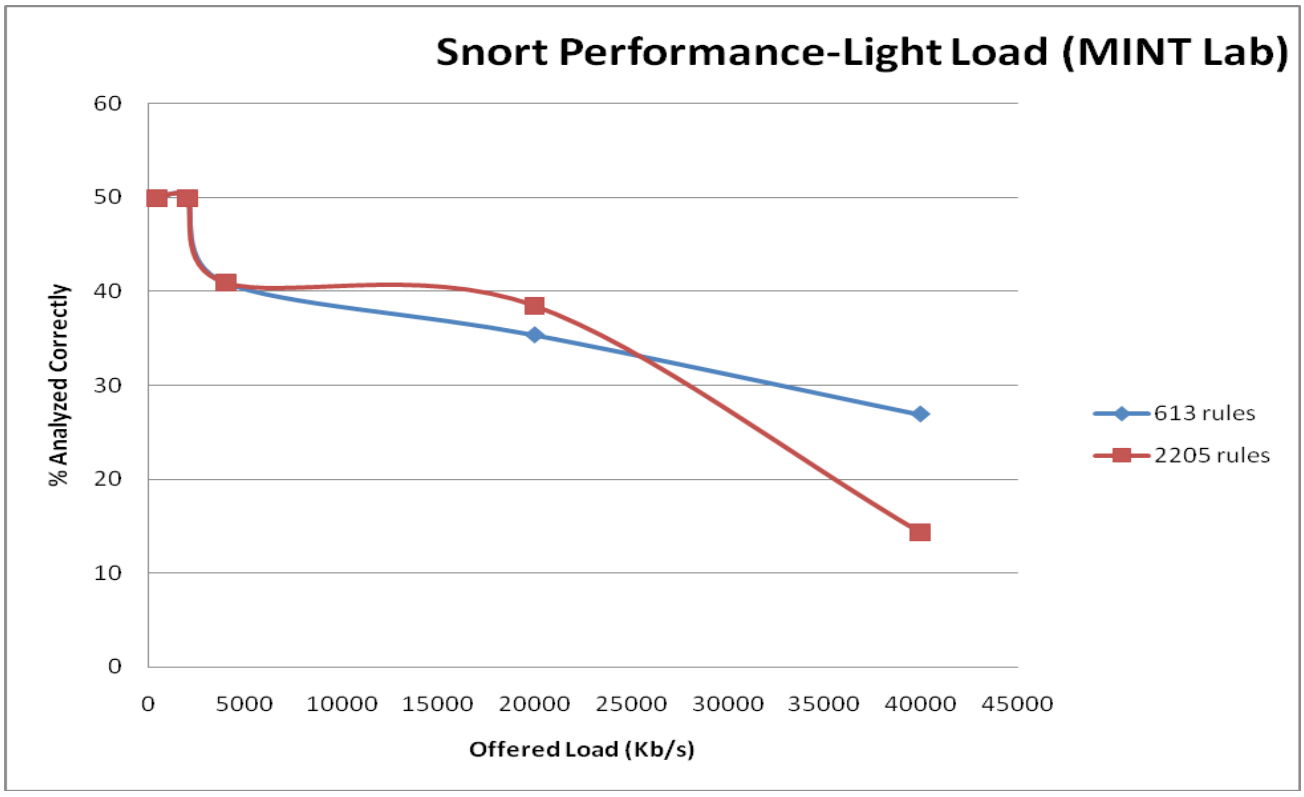
Table 7e: Constant Packet Rate of 100000/sec.

Number of Rules	Total packets	Packets Analyzed (%)	Packets Dropped (%)	Packets Outstanding (%)
1022	645916	0.63351891	98.73265254	0.633828547
2205	637365	0.278176555	99.43595899	0.285864458
3205	640766	0.146543356	99.69926619	0.154190453
4205	639889	0.012033337	99.96811947	0.019847192

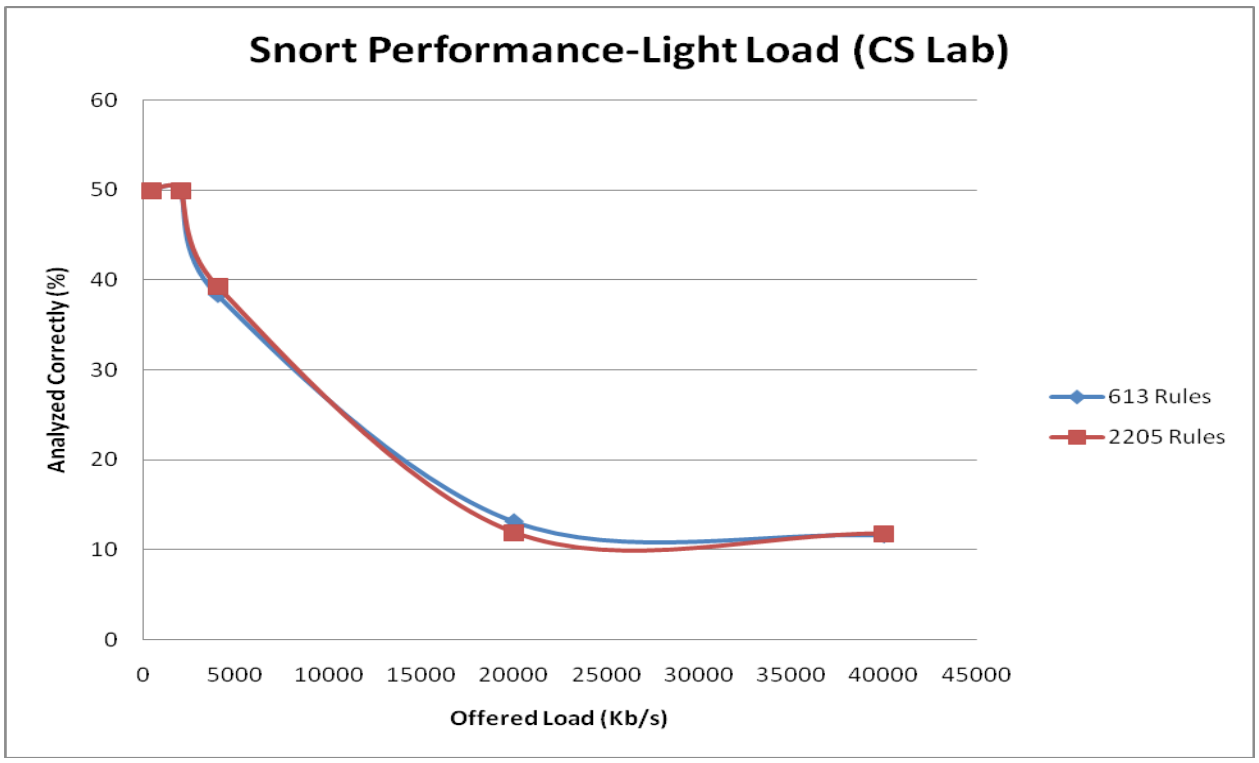
5.0 Analysis of Results

As stated above, this experiment was conducted in two different labs. The following graphs show the results for some selected experiments in these two labs.

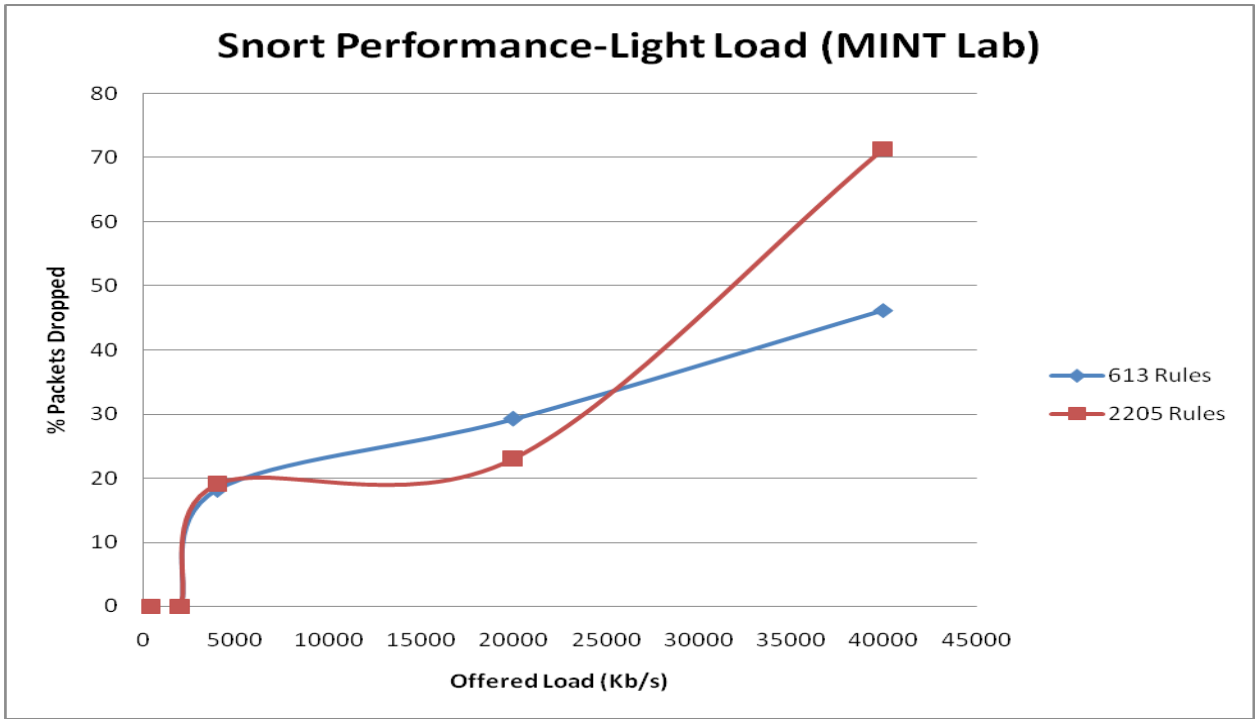
5.1 Performance of Snort under Light load



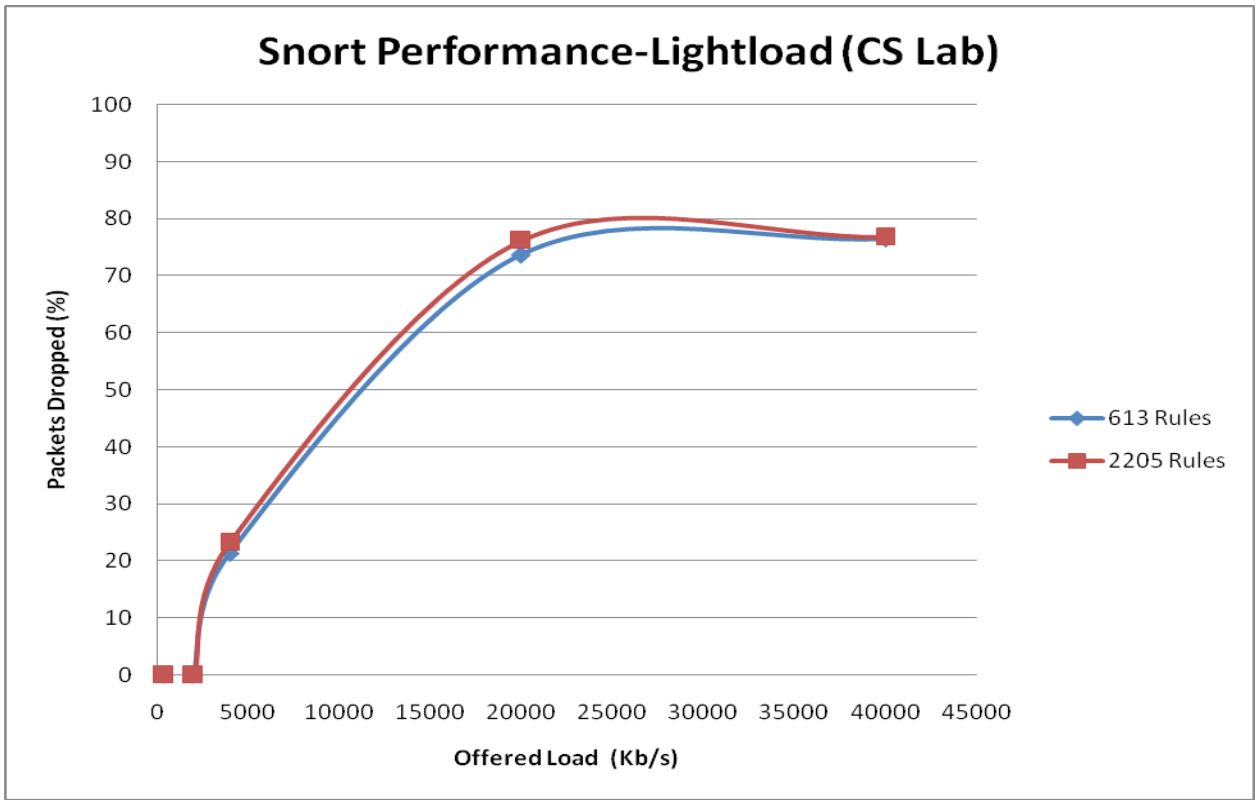
Graph 1 Snort Performance under light Load Conditions (% of Packets Analyzed Correctly)



Graph 2 Snort Performance under light Load Conditions (% of Packets Analyzed Correctly)



Graph 3 Snort Performance under light Load Conditions (% of Packets Dropped)



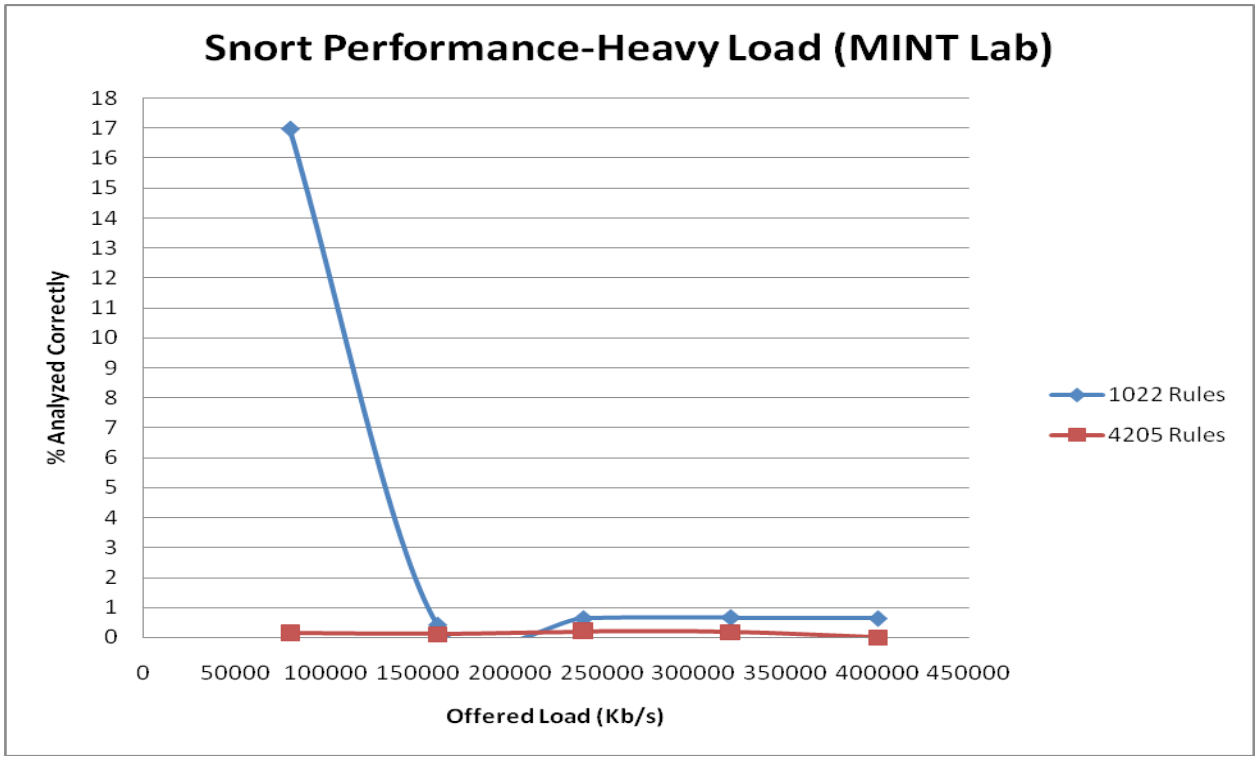
Graph 4 Snort Performance under light Load Conditions (% of Packets Dropped)

As can been seen from the graphs, the results in the two labs gave similar results. Comparing Graph 1 and Graph 2, it can be seen that Snort was able to analyze about 50% of the packets that it received. As the traffic intensity increased, this value decreased sharply to about 12% in both cases. But in terms of the number of rules, there was some slight difference in the two lab results; in Graph 1 (MINT Lab), it can be observed that as the number of rules increases more packets are being dropped. When the rules was at 613, the packet dropped to a rate of about 27% when traffic intensity was increased but when the rules was 2205, the rate dropped to about 12%. However in Graph 2 the number of rules did not play much of a role in the number of packets being dropped, as can be seen the two plots shows no difference in the number of packets dropped as traffic intensity was decreased. This difference may be attributed to that fact that in the CS lab the end point devices were much slower so the rate of packet generation was slow. Therefore, increasing the traffic intensity did not make much of a difference as the numbers of packets generation in a second remain constant.

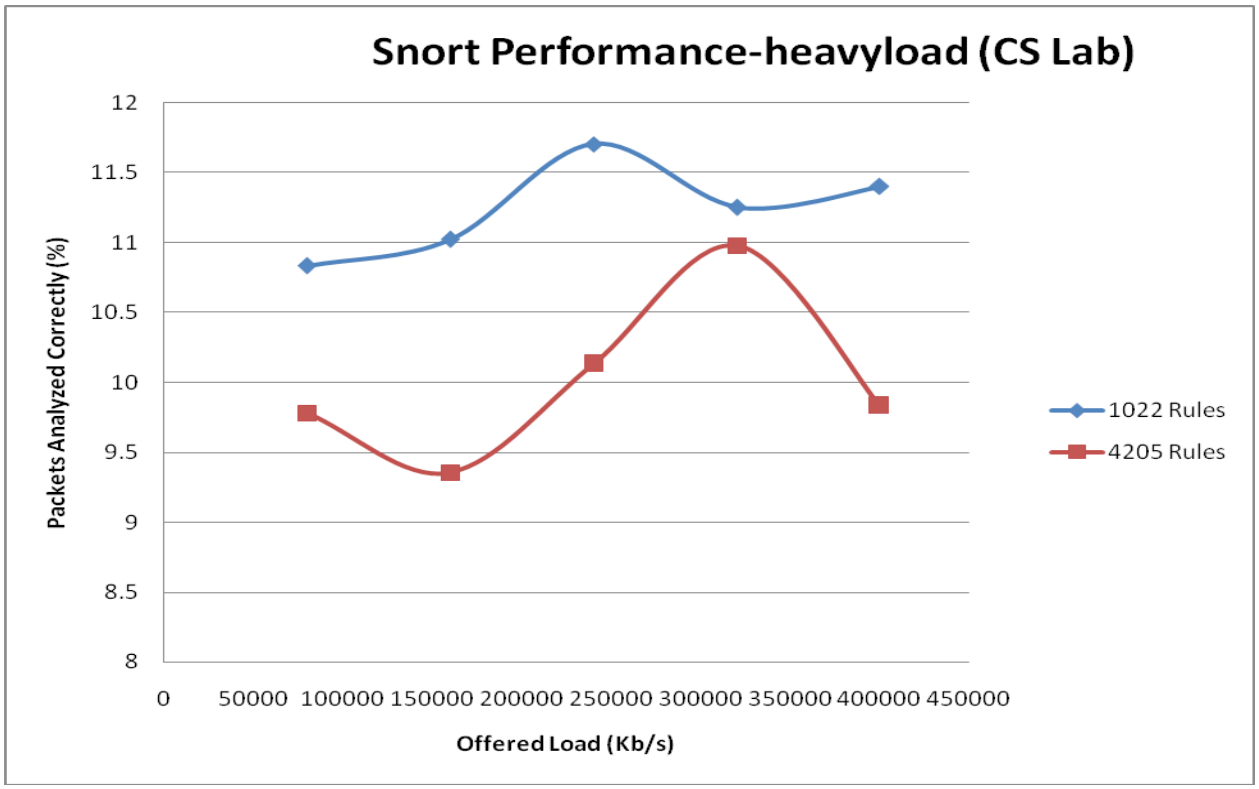
In terms of packets dropped, again the results follow similar trends with some slight differences. In Graph 3 (MINT lab), as traffic intensity increases, the packet dropped to a rate of about 71% when the number of rules were at 2205 and to about 47% when the number of rules were at 613. So here the number of rules played a key role in the number of packets being dropped as the load increase. However in Graph 4, the packets drop rate in both instances was about 79%.

5.1 Performance of Snort under Heavy Load

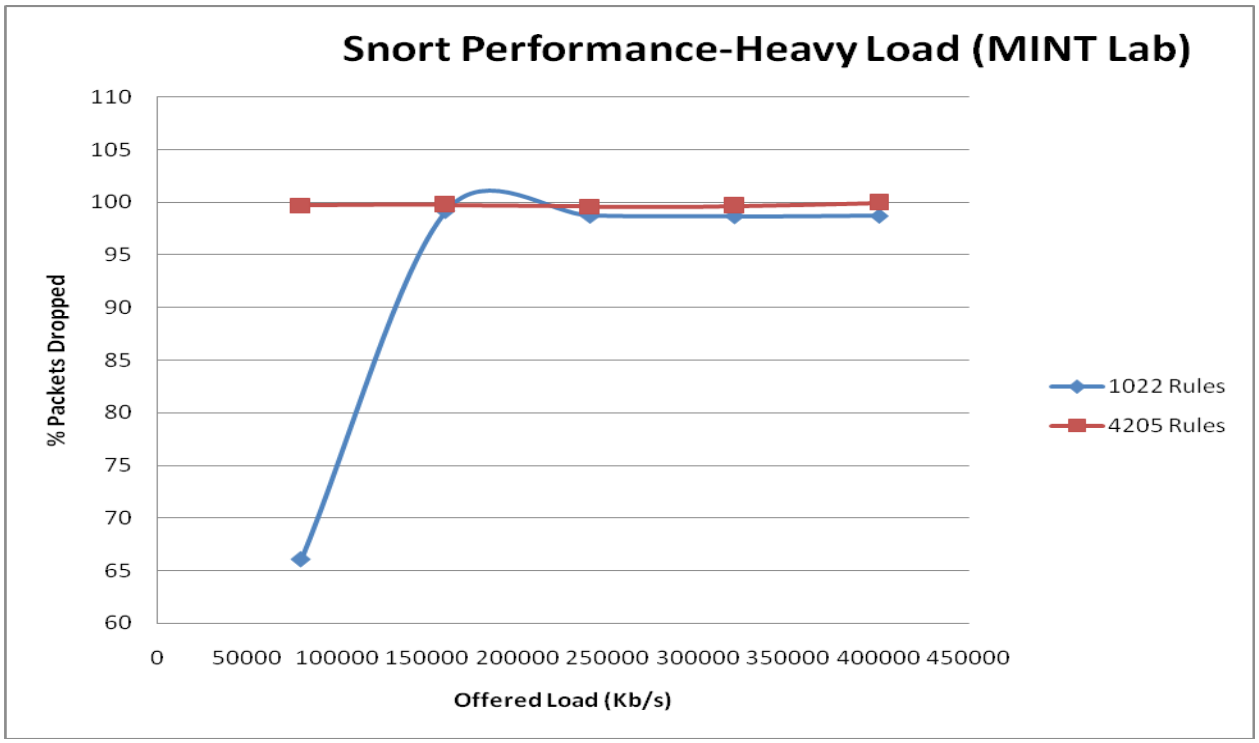
In this section the performance of Snort was evaluated under heavy load traffic condition with increasing number of rules. The following graphs show the results under this condition.



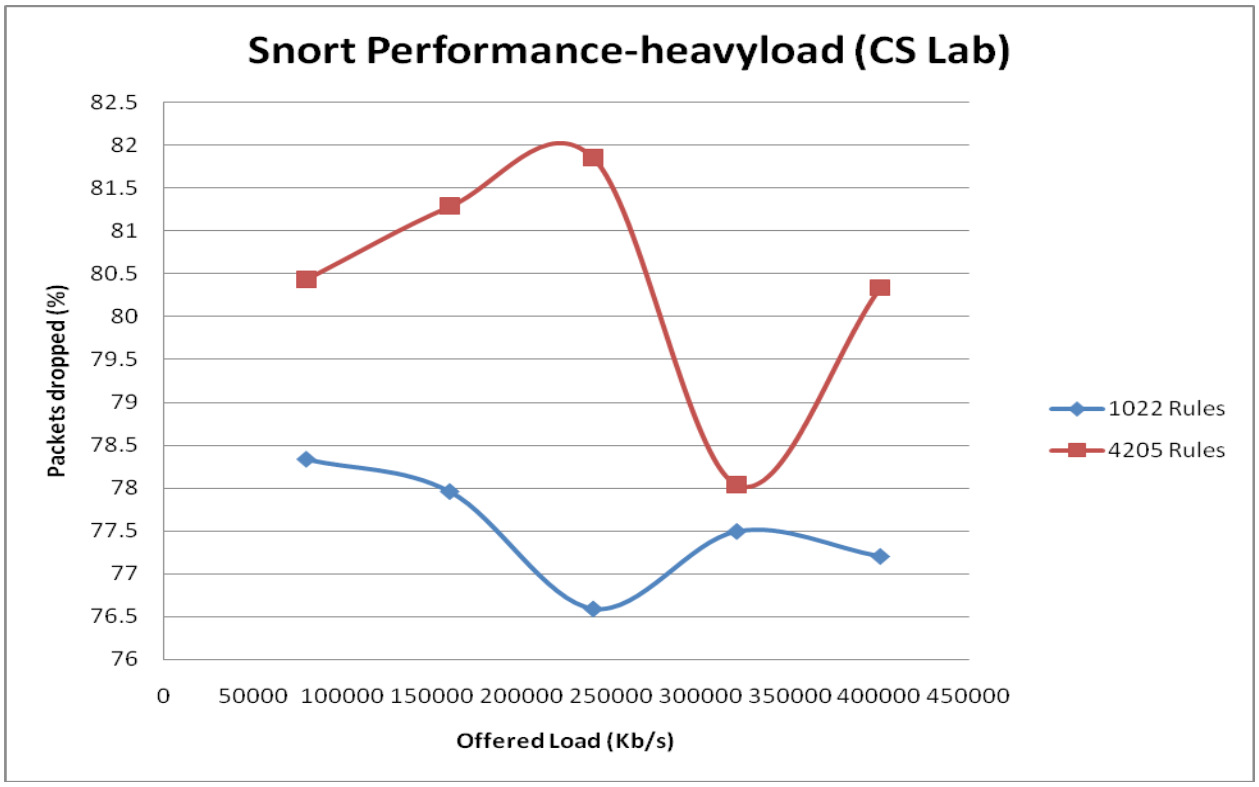
Graph 5 Performance of Snort under Heavy Load (% of Packets Analyzed Correctly)



Graph 6 Performance of Snort under Heavy Load (% of Packets Analyzed Correctly)



Graph 7 Performance of Snort under Heavy Load (% of Packets Dropped)



Graph 8 Performance of Snort under Heavy Load (% of Packets Dropped)

From Graph 5 and Graph 6, it can be seen that increasing the traffic intensity and the number of rules greatly affected the performance of Snort. Again there was some slight difference in the results from the two different labs. From Graph 5 the percentage of packets analyzed correctly almost fell to 0% (falling from 17% to almost 0%) but in Graph 6, the number of packets analyzed correctly fell to 9.2% (falling from about 12% to 9.2%). However, the common denominator in these results was that Snort performed badly in this scenario. The slight difference was due to the fact that the number of actual packets generated in the MINT lab was about twice the amount generated in the CS lab due to the end device capabilities. Here too the number of rules played an important role in the number of packets analyzed correctly. In graph 5 when the number of rules was at 1022 the number of packets analyzed correctly fell from 17% to almost 0.5% but when the rules was at 4205, the number of packets analyzed correctly constantly stayed at almost 0%. A similar pattern was observed in graph 6 where the graph of where the

number of rules was at 1022 always stayed on top of the graph where the number of rules was at 4205.

In terms of packets dropped (Graph 7 and Graph 8), the trend is similar in the two labs. As can be seen the number of packets dropped greatly increased as the traffic intensity was increased. The drop in graph 7 almost rose to 100% while in graph 8; the drop was about up to 82%. Again the difference was due to the fact in the MINT lab more packets were been generated per second.

In the CS lab statistics was also collected on the memory and CPU usage. The following bar chart shows the average usage of memory and CPU by Snort during the experiment in both the light and heavy load scenarios.

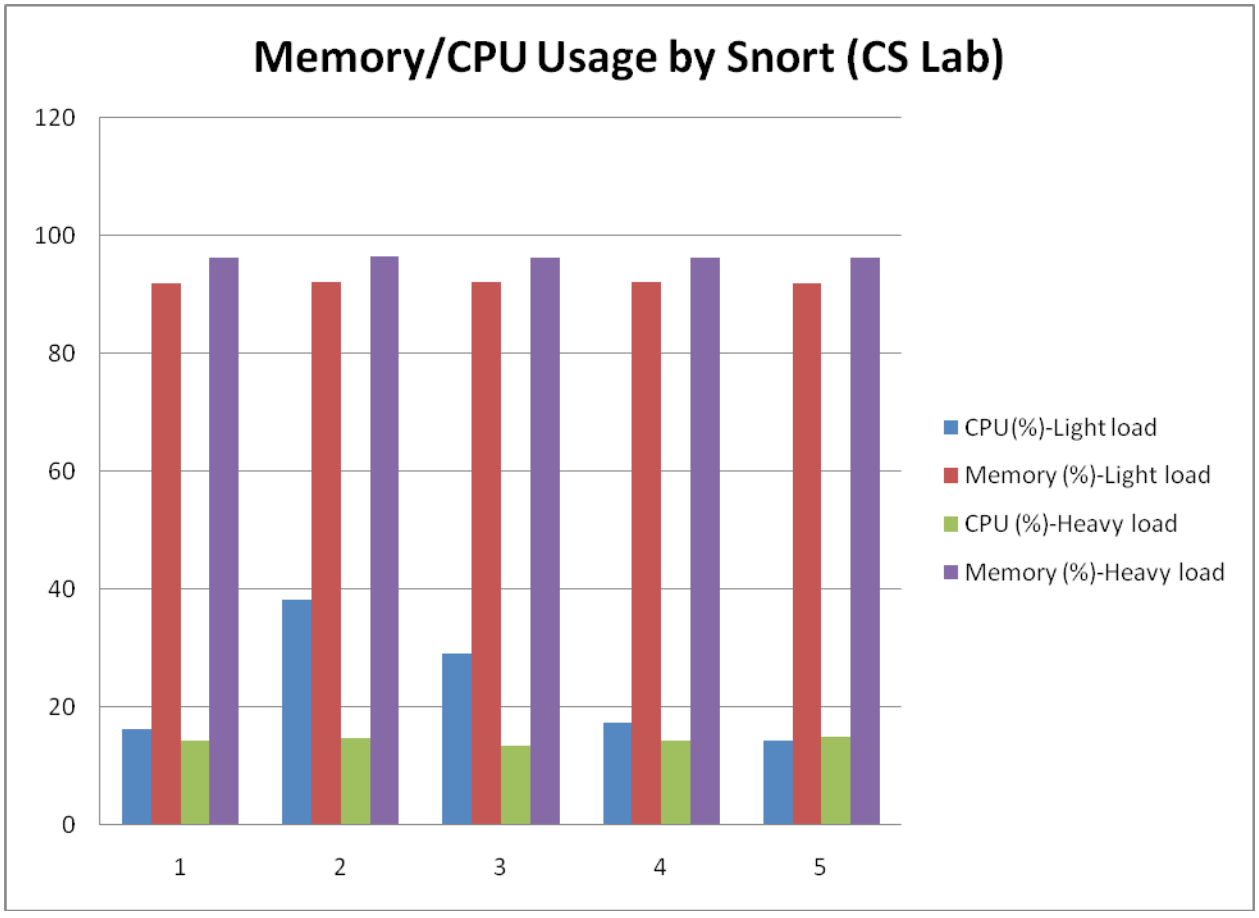


Figure 5.1 Percentage of Memory/CPUUsage by Snort (CS lab)

As can be seen the average CPU usage in the light load scenario was even higher than the heavy load scenario. This can be attributed to the fact that in the heavy load scenario, Snort was actually dropping almost all of the packets, so it was not doing much work. However the average memory usage was extremely high almost hitting the 100% mark. (It was observed that with an increase in the number of rules, the CPU usage was almost up to 100% when Snort was starting or initializing.)

6.0 Conclusions and Recommendations

In the experiments conducted, the performance of SNORT was evaluated under different traffic intensities with different end host devices in two different labs. In these two labs, the performance of Snort was highly ineffective when traffic intensity was increased. In the MINT lab where the end host had more packet generating capabilities, the rate of packets dropped was almost 100% which meant Snort was almost allowing every packet to pass without applying the rules to those packets. However, in the CS lab where the packet generating power of the end host devices were limited, the performance of Snort under heavy load traffic was slightly more effective with a maximum packet drop rate of about 81%.

Under light load conditions, Snort performance was much better with packet drop rate going from 0% to about 80%. However it was observed here that although there was a 0% drop rate, Snort was actually able to apply all the rules to about 50% of the packets and reported 50% of the packets as outstanding.

As stated in the introduction, the main aim of these experiments was to investigate the behavior of Snort under two scenarios. The first being that under high load conditions, Snort drops packets without informing the network administrator and the second being that under high load conditions, Snort allows packets to pass which violates one or more rules in the rule set.

In the experiments, there was no evidence found to support the first claim, however the second claim was verified. In the experiments it was found that as Snort is overloaded with traffic that it cannot handle, it allows the packets to go through without applying any

rule to the packets. This case was further investigated using wireshark at the receiver end device and creating a rule that was to drop a certain flow (i.e. tcp) and running Snort in the inline mode. Although Snort was to drop all of the flow, not all of them were dropped as some of the packets were captured by wireshark at the destination device.

Based on the results of these experiments, using a machine with more CPU power and sufficient memory to improve the matching abilities of it, is recommended Load-balancing can also be used by installing Snort on multiple devices and directing traffic to these devices where Snort is installed. In this way the load on Snort will be much less and the performance of Snort will improve.

7.0 References

[1] http://en.wikipedia.org/wiki/Intrusion_detection

[2] WIND: Workload-Aware Intrusion Detection

Recent Advances in Intrusion Detection, Proceedings Lecture Notes in Computer Science 4219: 290-310 2006

[3][4] WIND: Workload-Aware Intrusion Detection, Page 291

[5] WIND: Workload-Aware Intrusion Detection, Page 302

[6][7] WIND: Workload-Aware Intrusion Detection, Page 304

[8] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Information Security Applications Lectures in Computer Science
3325: 188-203 2005

[9] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 191

[10] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 194

[11] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 195

[12] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 195

[13] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 196

[14] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 197

[15] Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading, Page 201

[16] Snort Offloader: A Reconfigurable Hardware NIDS Filter
Field Programmable Logic and Applications, 2005, International Conference on 24-26 August 2005, Pages 493-498

[17] Snort Offloader: A Reconfigurable Hardware NIDS Filter, Page 493

[18] Snort Offloader: A Reconfigurable Hardware NIDS Filter, Page 494

[19] Snort Offloader: A Reconfigurable Hardware NIDS Filter, Page 497

[20] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 40

[21] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 41

[22] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 43
[23] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 44
[24] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 47
[25] Snort IDS and IPS Toolkit, Syngress Publishing Inc. 2007, Page 302