Florida State University
College of Arts and Sciences

Introducing Multi Threaded Solution to Enhance the Efficiency of Snort

By

Syed Yasir Abbas

December 07, 2002

A project submitted to the Department of Computer Science for the degree of Masters of Science in Computer Network Systems Administration

Project Committee Members

Dr Lois W. Hawkes (Major Professor)

Dr Xin Yuan

Dr Alec Yasinsac

Abstract

In these days where the whole world is connected through Internet, Intrusion Detection Systems have become an essential part of a network. Where security is the major concern of any organization, performance is also a crucial factor. This project is an attempt to look at a fast growing Intrusion Detection System, Snort, and find out ways to improve the overall performance by introducing threads.

Acknowledgment

I am very grateful to Dr Lois W. Hawkes for her guidance and patience with me and my project.

I am also thankful to Jed Pickel (jed@pickel.net) from Snort community for his kind suggestions.

Thanks to the fault tolerance team, especially Shayne Steele, Srikanth Ravula and Tavaris Thomas for suggestions.

This Project was done at SAIT, Security and Assurance in Information Technology Lab at the Computer Science Department, Florida State University. Thanks to Dr Alec Yasinsac for allowing me to do it over there.

Special thanks to Dr Xin Yuan. All the concepts and expertise about threads and unix programming was acquired from the classes taken under him.

Table of Contents

Abstract ***2***

Acknowledgment ***3***

Objective: ***5***

Background ***6***

Problem Definition: ***20***

Project Design: ***21***

Implementation of multi-threaded Snort: ***23***

Data Structure of the multi-threaded Snort: 24

Functions for multi-threaded Snort: 27

Performance: ***29***

Results: ***33***

Conclusion: ***59***

Future Work: ***60***

References: ***61***

Appendix A: Sample Status file ***63***

Appendix B: Testing Program ***66***

Appendix C: Project Code ***68***

Appendix D: Introducing Levels in Snort – Potential Project Overview ***78***

Objective:

Enhancing the performance of Snort, an Intrusion Detection System:

- This project is an attempt to contribute to the field of Network Security by improving the performance of an Intrusion Detection system with a fast growing popularity. Snort provides the users with flexibility and with open source; it lets the advanced user contribute to the Intrusion Detection Library.

- The goal of this project is to generate faster responses by decreasing the burden of the main program. The current situation is that the Snort process itself has to deal with generating messages for detected intrusions. Whenever high disk latency overheaed and network delays are involved, that makes it slower. With the addition of concurrent programming with threads for output tasks, this project shows an improvement in the Snort system performance.

Background

Intrusion Detection System:

An Intrusion Detection System dynamically monitors the actions taken in a given environment, and decides whether these actions are symptomatic of an attack or constitute a legitimate use of the environment.

Kumar and Spafford [94] have defined it as:

Intrusion Detection is primarily concerned with the detection of illegal activities and acquisition of privileges that cannot be detected with information flow and access control models.

As the Internet develops, it has been realized that security precautionary methods such as firewalls are not sufficient to find out when a network is under attack. What is needed is a three-fold process:

> First step is to detect whether there is an attack or not.
> Secondly it should collect forensics so that the guilty may be punished.
> It should also collect statistics to find out how effective or ineffective the current security measures are.
> Finally it should gather information about the attack to understand its behavior.

Intrusion Detection is divided into two main categories:

Knowledge-base Intrusion Detection systems:

This kind of Intrusion detection systems maintain a database of known signatures for attack and match those to the current data to figure out whether there is an attack going on.

Behavior-based Intrusion Detection Systems:

This kind of Intrusion Detection System has an exclusive nature, that is, to deny all activities except those that fit in under pre-defined patterns. A database of permitted activities is maintained and any behavior seen otherwise is dubbed suspicious thus setting off the alarms. This can be used in sensitive peripherals that require more strict security and would prefer limited activities as compared to Knowledge-base Intrusion Detection Systems that only report a possible intrusion if some activity matches known attack patterns. This approach is also useful for employers suspicious towards their employees.

Basic Intrusion Detection System Model as proposed by Ilgun, Kemmerer [95]:

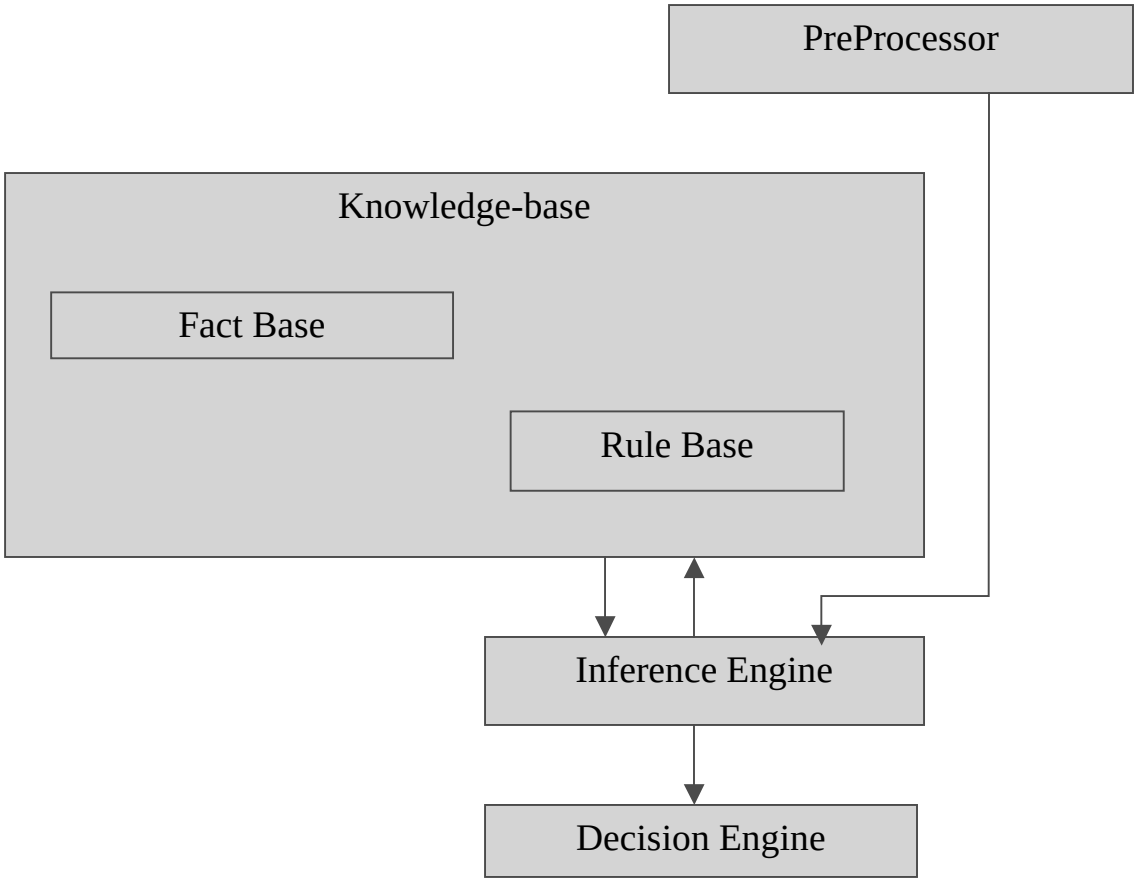


Figure 1: IDS Model

Preprocessor is responsible for collecting the data and translating it into a format acceptable to the Inference Engine.

Knowledge-base contains the signatures and rules for known attacks. Fact-base contains facts, rule-base contains rules that apply to those facts.

Inference Engine collects data from the preprocessor, matches those with the facts and rule base and applies rules to match. Inference Engine is also supposed to update the fact-base as new attacks are known.

Decision Engine is responsible for responses on the basis of the inference earlier taken.

Host-based Intrusion Detection Systems vs Network-based Intrusion Detection Systems:

HIDS (Host-based Intrusion Detection System) is installed on a single host and it checks for possible intrusion on its host only. So the monitoring range is limited to the host machine only.

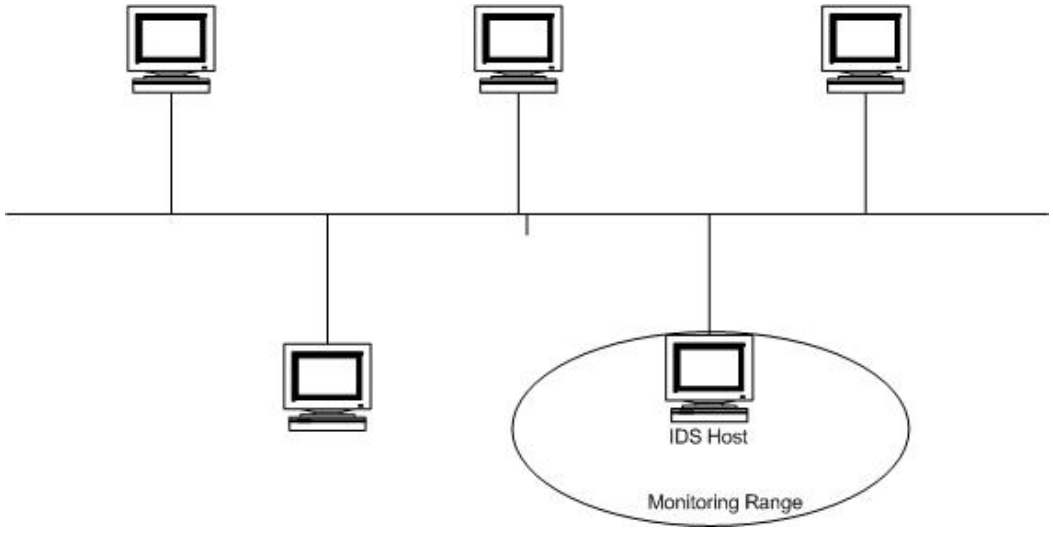


Figure 2: Host Based IDS

NIDS (Network Intrusion Detection System) resides on a single host and sniffs (monitors) the packets on a network level. So it checks for possible intrusion on the whole network.

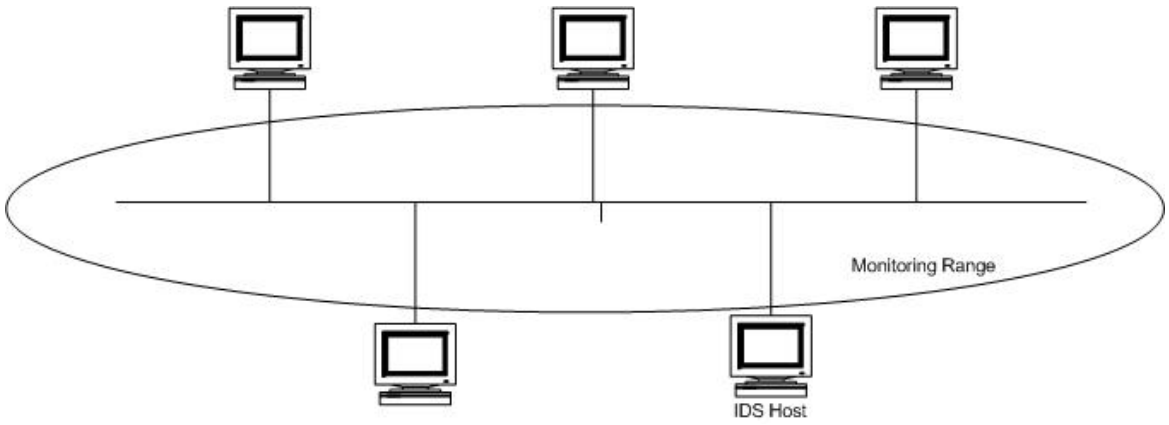


Figure 3: Network IDS

Intrusion Detection System (IDS) Products:

Some of the well-known Intrusion Detection Products out in the market are:

- Network Flight Recorder

NFR is one of the most popular, if not THE most popular, commercial Intrusion Detection systems. It has two different versions: Host-based and Network-based. The rule writing in NFR is done in N-code, which is a whole language. [NFR Security, 2002]

- Cisco IDS

Cisco had their Intrusion Detection system earlier named as NetRanger but with newer version they dropped that name. It provides both Host-based and Network-based protection. Cisco has divided their IDS in two parts: Sensors to detect the attack, and Director to figure out what to do after that. [Cisco Systems, 2002]

- RealSecure from Internet Security System

They divide their product in to a 3-tiered architecture: Management layer, Sensor layer and Event Collector. Event Collector lets the Sensor layer only worry about monitors and takes care of the data stream and database syncs. Rule writing is distributed between different .ini and text files. ini files have initialization values and parameters to be used by the Management layer to create text log files. Other text files include information like version and help url. [Internet Security Systems, 2001]

- Snort

Snort is a lightweight intrusion detection system with a rapidly increasing popularity. Snort is available free and open source. It is a network-based intrusion detection system. Rule writing for Snort is very easy and that is one of the reasons why Snort has been so quick in responding to the new attacks. Snort applies the Knowledge-base Intrusion Detection systems approach. But it can be configured to deny all activities except few to apply the Behavior-base Intrusion Detection systems approach. [Roesch, 1999]

Snort:

As defined in the famous “Lisa Paper”[99] by its creator, Martin Roesch himself, Snort is:

> a cross-platform, lightweight network intrusion detection tool that can be deployed to monitor small TCP/IP networks and detect a wide variety of suspicious network traffic as well as outright attacks.

Snort is under GNU General Public License (GPL) and it can be downloaded free of cost. It started as a student project of Martin Roesch. Over a period of time, it has been built through the contributions of people from different parts of the world over the mailing lists.

Why was Snort chosen for this project?

- Open Source

Snort is under GPL license and thus the code for Snort is open. It has been developed over time by the contribution of many people.

- Faster emergency response

Snort has an easily managed Rule base. So for every new attack, the rule writing is easier and thus the Snort community comes out with a faster response for users than other such systems.

Comparison Between Snort and Network Flight Recorder (NFR):

Snort is only a NIDS whereas NFR is both NIDS and HIDS. NFR sensors require dedicated hardware whereas Snort NIDS can be installed on any machine. Snort is still developing protocol coverage (has TCP, UDP, ICMP, ARP). NFR, as compared to Snort, covers a wide variety of protocol coverage, for example DNS and RSH protocols. Snort is evolving towards being more user friendly with graphical representation, whereas NFR is all graphical based. Though Snort rule writing is easier, NFR rule writing requires learning a whole new language. Signature coverage is the same except that NFR covers more network protocols.

Below is a comparison between Snort and NFR given by http://zen.ece.ohiou.edu/~nagendra/compids.html .

Name of IDS	**NFR NID**	**Snort**
Type	Network-Based	Network-Based
Network speed & Over head	NID 200 - 100MBPS NID 100 - T3 High speed networks	Moderately speed networks
Threat signature language	N-code language "full featured scripting language"	Snort rule - not as complete as n-code
Method (s) of detection	Misuse/Anomaly	Misuse Detection Snort is moving towards anomaly detector
Portability	Needs dedicated machine	Runs on almost any system

Real-time Operation	yes	yes
Attack resistance	yes	No, But can act as Honey pots (Honeypots explained on next page)
Detection time	Fast	Moderate
IP Defragmentation	yes	yes
TCP stream reassembly	yes	yes
Working	Central	Central
Open Source	No	Yes

Honey Pots:

Honeypots are networks with intentionally introduced vulnerabilities in order to attract hackers. One reason for this is to divert a potential attacker's attention from the actual network. Another reason is to monitor those attacks.

Snort can be used as a honeypot by being installed in such a network, so that it records the possible intrusions made by hackers to figure out what attack are generally popular and whether there are exploits generally overlooked by network administrators

Snort Flow:

Standard Snort starts as a single program. It creates one thread each for each network interface.

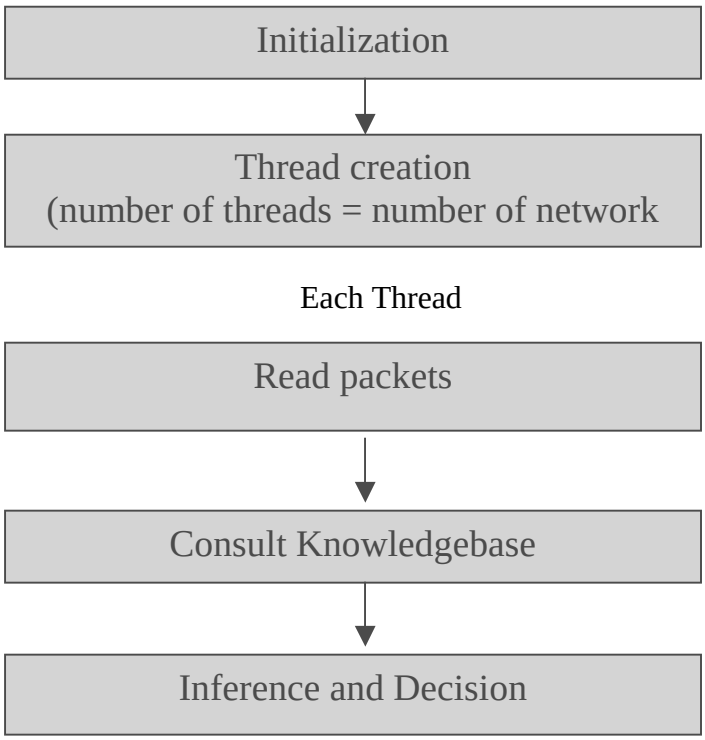


Figure 4: Snort Flow

Snort Use:

There are three ways Snort can be used:

- Sniffer
- Logger
- IDS

Snort as a Sniffer:

Snort can be used as a simple Sniffer, that is, it can be used simply to monitor the network traffic. User can choose between headers, headers and data, and header and data with more description.

Snort as a Logger:

Snort can be used to log all the packets into a file. This is one step ahead of the sniffer as a logger logs the packets into a file for selected traffic. It can log in simple text or in binary mode as tcpdump format. The user can specify the home network. It also lets the user log only control packets by filtering in icmp packets only.

Snort as an IDS:

This is the full mode operation of Snort. Along with both of the above options, an important command line option is the name of the file that contains **rules** that need to be checked against packets. The procedure below is followed:

Procedure:

- Network packets are sniffed.
- Those packets are matched against the patterns given by the rules in one or more file.
- If there is a match, an action is taken as determined by the corresponding rule.

A sample snort command would be (as in the manual):

./snort -dev -l ./log -h 192.168.1.0/24 -c snort.conf

This tells the user to run snort displaying the full packets on screen, having 192.168.1.0/24 as the home network, and to use snort.conf as the rule file. Snort can also be run as a daemon in the background that will be generating alerts without showing the packets on the screen.

Actions:

Possible actions that can be taken when packets are matched against known patterns are:

- Alert
  - Snort generates Alert messages for detected Attacks. (See following section for detail)
- Log
  - Another possible action is to generate Logs for Packets. The logs can be either saved into a human readable text file, or in binary tcpdump format
- Ignore
  - This option basically tells Snort to do not do anything for the matched pattern.

Other Action Types:

- Activate
  - Activates Alerts are those types of alerts that, when matched, turn on another dynamic rule
- Dynamic
  - Dynamic rules, when turned on by an activate rule, act as a log rule.

User defined Rule Types:

- Users are allowed to create new types of rules.
- Users can specify:
  - Type (alert / log / ignore)
  - Log File
  - Output Database (through a db plug-in)

Alerts:

Alerts are generated whenever an attack is detected (Detecting an attack refers to some packets matching the user pre-defined rules in Snort rule file.)

Alert mode:

Fast

- This mode generates the alert with only a brief description of the packets.
- It is suited for the environment prone to generate more false alarms or where attacks are obvious and a brief description of the packets is enough to understand the attacks.

Full

- This mode generates the alert with full packet logging.
- It is suited for the environment where attacks seldom happen or Packets have to be studied in detail to research the attack.

Unsock

- This mode directs alerts to a Unix socket.
- Unix programs can be written to use the alerts in a customized manner.

None

- This mode turns off alerts.
- It can be used for testing purposes.

Snort Rule File:

Snort rule file can contain, along with rules in snort rule format, some include directives that let other files be used as Snort rules. Snort comes with some pre-written rules. User can write his/her own rules customized to the environment where Snort is going to be used.

Alert output Messages:

Alert output messages are directed towards one of the following pre configured modes. [Roesch, Green, 2002]

- Syslog
  This sends Snort alerts to the syslog facility, as per defined by the host operating system.

- Server Message Block:
  This will direct Snort Alert messages to appear as Windows Pop-up messages on a windows machine running netbios.

- TCP dump format:
  This will log Snort alert messages in a tcpdump format file.

- Fast_alert
  This will print Snort alerts with quick one line format in a file.

- Full_alert
  This will print Snort alert messages along with the packet headers for which the alert is being generated.

- XML
  This will log Snort messages in a Simple Network Markup Language using the DTD that comes with the Snort package.

- Database
  This mode lets Snort messages to be stored in a SQL database

- SNMP Trap
  Using this mode, Snort alerts are sent as a SNMP (Simple Network Management Protocol) trap to a network management system.

- CSV
  Snort messages can be logged in CSV (comma-separated values) format that is importable to a database.

- Unified – Binary Fast

Simple Rule format:

Action Protocol SourceIP(s) SourcePort(s) Direction DestinationIP(s) DestinationPort(s) content_to_match message

Example:

Alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg: "mountd access";)

From this rule, we obtain the following information:

Protocol	TCP
Source IP address	any
Source Port	any
Destination IP address	192.168.1.0 / 24
Destination Port	111
Pattern to match	|00 01 86 a5|
Alert message to generate	mountd access

Threads:

A thread is defined as the basic unit of CPU utilization. [Silberschatz, Patterson, Galvin, 1991]. Threads are also known as lightweight processes. Processes have a separate data space whereas threads within a process share most of the data. Where context switching for processes requires copying the data back and forth into the memory, threads do not require much of it.

Threads within a single process share the following:[Stevens, 1998]

Global data
Process Instruction
Open file descriptors
Signal handlers
Current working directory
User ID
Group ID

Each thread has its own:

Thread ID
Set of registers (Program Counter, Stack Pointer etc)
Stack (Local variables and Return Addresses)
errno
Signal mask
Priority

Concurrent Programming:

Many processes can be multi-tasked on a single CPU. This provides many advantages, one of which is computation speedup. That is our main concern for this project.

For a single program, Concurrent Programming can be provided using two different methods:

- Forking more processes
- Creating many threads

Threads are developed to improve performance. Therefore the hypothesis was that the use of threads for output tasks would also improve the performance of Snort.

Problem Definition:

Introducing Threaded output plug-in

Alert Output is taken care of by the same Snort Process that is doing everything else. This becomes a bottleneck when the disk latency is high or where Snort alerts are being sent to another machine and network delays are involved.
So the objective of this project was to introduce threads that will take care of the output while the main process will keep detecting attacks.

As of now, the Standard Snort process itself has to take care of writing alerts to the proper places.
Letting another thread take care of it will make it faster for the output plug-ins to put alerts into action.

This Project makes changes in the Standard Snort system by using the concurrent programming technique and relieving the main process of the output management. There can be two different methods to do that:

- Forking more processes.
- Creating more threads.

Forking child processes requires heavier context-switching that involves copying the data to the child's memory space. That is expensive. Also the inter-process communication requires establishing pipes or sockets. So threads were chosen as a solution. Thread creation is much faster than forking processes. Also the inter-process communication is simple and efficient as all threads are sharing data.

Project Design:

There will be one or more threads created to deal with the output generated by the alerts.

For each of the threads in the multi-threaded solution, Snort Engine writes alerts to a shared queue and then carries on with its normal execution. The output thread reads alerts from the queue and takes care of the alert messages.

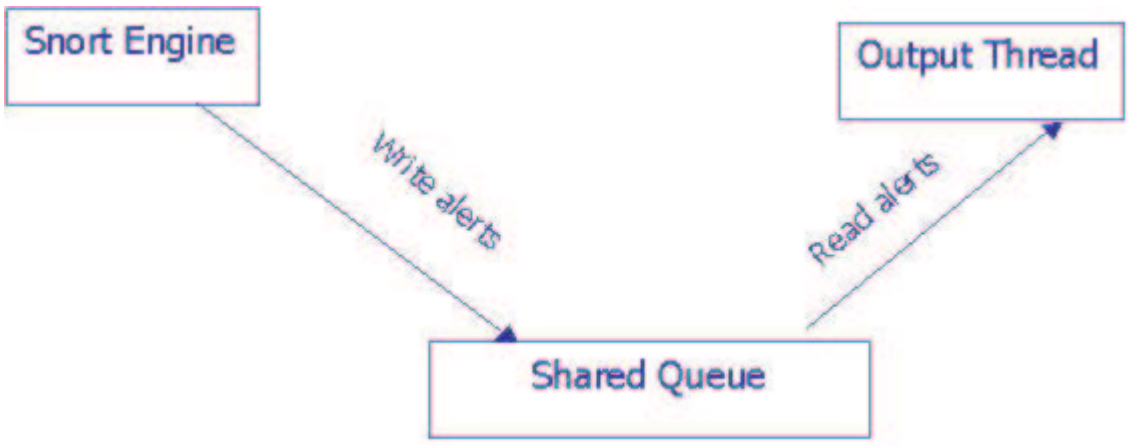


Figure 5: Single Thread solution

For n Threads, the number of threads is determined by the Snort user (typically a network administrator) with the command line argument of Snort. Snort Engine is not concerned with the number of threads. It will just write the output to the shared queue and carry on. The Output Threads take turns taking care of the alert messages.

If the count of alert is i, then thread j is responsible for alert number (i MOD n), where n is the total number of threads.

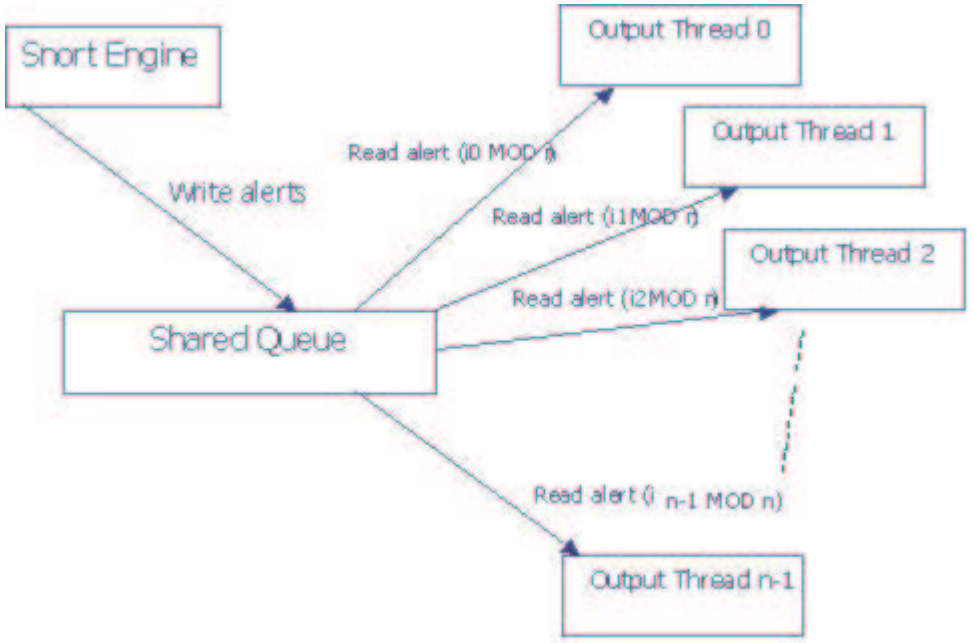


Figure 6: Multi Thread solution

Implementation of multi-threaded Snort:

Work Environment:

The small 192.168.1.0/24 network has three computers. One has Linux workstation and the other has Windows NT workstation. The third computer is a Linux Server and that is where Snort is installed. All three computers are connected through a hub and outside connection is optional for the server.

The work environment was setup at SAIT, Security and Assurance in Information Technology Lab at Computer Science Department, Florida State University.

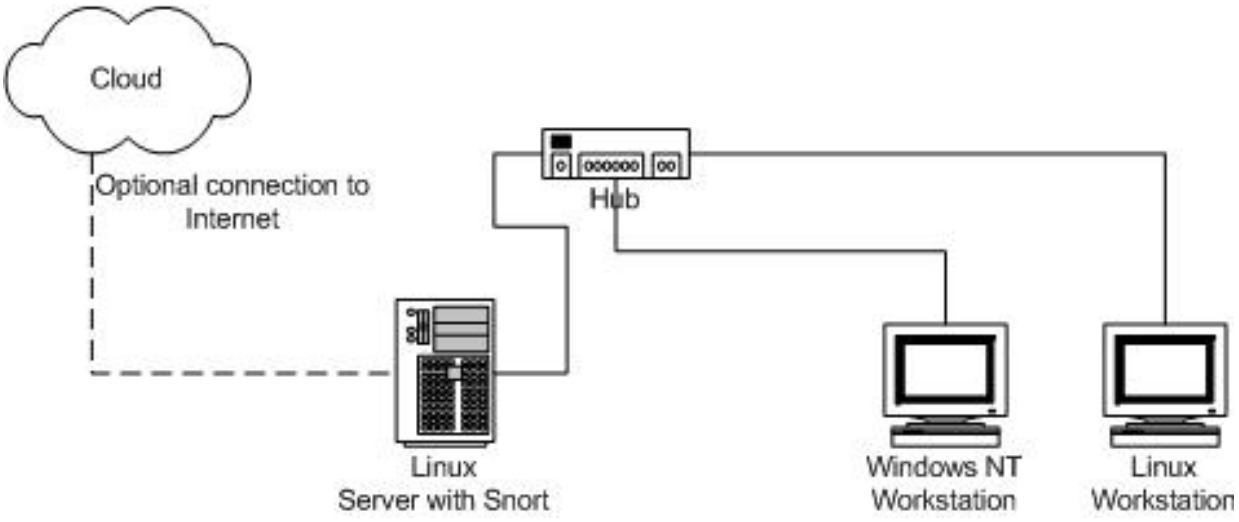


Figure 7: Work Environment

Data Structure of the multi-threaded Snort:

Data Type	Variable Name	Description
	NO_T	defined in multop.h to have the maximum size of threads array. It is pre defined as 100.
int	num_of_threads	obtained from the command line argument (that is used to run Snort by the Snort user) to indicate the number of threads to be created.
pthread_t	outputthread[NO_T]	the array of threads that will be created when Snort starts. The number of actually created threads is limited to num_of_threads.
pthread_attr_t	tattr[NO_T]	Thread Attributes associated with each thread.
pthread_mutex_t	mx[NO_T]	array of mutexes associated with each thread to see whether it is busy or not.
pthread_mutexattr_t	mxattr[NO_T]	array of mutex attributes associated with each mx.
pthread_mutex_t	mx2[NO_T]	array of mutexes associated with condition variables for each thread.
pthread_mutexattr_t	mx2attr[NO_T]	array of mutex attributes associated with each mx2.
pthread_mutex_t	itsme	mutex for status file.
pthread_mutexattr_t	itsmeattr	mutex attribute associated with mutex itsme.
pthread_cond_t	condts[NO_T];	array of condition variables for each thread.
pthread_condattr_t	condatts[NO_T];	array of condition attributes for each condition variable.

Data Type	Variable Name	Description

struct thrarg {
int myrank;
};
		Each thread is assigned a rank which is passed to it when being created.

struct argsthread {
Packet * p;
OptTreeNode * otn;
Event *event;
};
	struct argsthread *args1[NO_T];	the data structure required by the Alert
		
		Action. This is the data structure where data is copied before calling the thread.

FILE *	fdmtcheck	- File pointer for status file.

The correspondence is between arrays of threads, mutexes mx and mx2, condition variable array condts and the struct argsthread. For each thread that is an element of the array Outputthread, there is one mutex mx, one mutex mx2, one structure args element, and one condition variable condts. Logically, we can say that it is a table like:

Outputthread	mx	mx2	condts	args1
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
.	.	.	.	.
.	.	.	.	.
n-1	n-1	n-1	n-1	n-1

The correspondence is like this:

Outputthread – mx:

mx[i] is the mutex that is tested to see whether Outputthread[i] is busy or not.

Outputthread – mx2:

mx2[i] is the mutex that is associated with the condition variable condts[i].

Outputthread – args1:

Outputthread[i] looks for the data from args1[i].

Outputthread – condts:

condts[i] is the condition waited upon by Outputthread[i] .

Number of threads:

num_of_threads variable is given by the Snort user, typically a Network Administrator, through the command line argument of Snort. This project has added a –Y option to the Standard Snort system to allow the user to configure the number of threads.

Functions for multi-threaded Snort:

Functions Modified from the Standard Snort:

main:

main function had to be changed from the Standard Snort so that it calls the mtinit to take care of the initialization.

AlertAction:

Instead of calling the functions that will take care of the alert messages, this function now:

- Copies the data structure to be used by threads.
- Tries lock to a mutex corresponding to a thread to figure out whether that thread is currently busy or not.
- After getting the rank for a free thread, tells the thread to start its execution by signaling through the condt condition variable.

ParseCmdLine:

To introduce the '-Y *n*' argument for number of threads.

New functions:

void outthreadfunc(void *);

This is the function that serves as the start_routine for threads.

- This function waits for the signal from the main program.
- After getting the signal, it locks a mutex to show that it is busy.
- It takes care of the alert.
- It locks a mutex to write into the status file every 25 alerts.
- It unlocks the mutexes it held and again goes into wait state.

void lockagain(int);

This function takes care of handling the SIGUSR2.

void mtinit(void);

The initialization function called by the main. This function:

Initializes:

- Thread attributes

- Mutex attributes
- Condition attributes
- Threads
- Mutexes
- Condition variable

Creates:

- all the threads

Allocates the memory to data structure and
Locks the mutexes.

The code for the functions appears in Appendix C.

Performance:

Expectations:

The process of generating alerts should get faster with Threads taking care of handling Alerts wherever network delays and disk latency is involved. The idea behind multi-thread programming is to speed up and improve performance. In the multi-threaded solution, output generation for Alerts is taken care of by the threads, and this should make the multi-threaded Snort faster than Standard Snort for heavy load for disk access and network, where the main process itself was responsible for output overhead.

Testing mechanism:

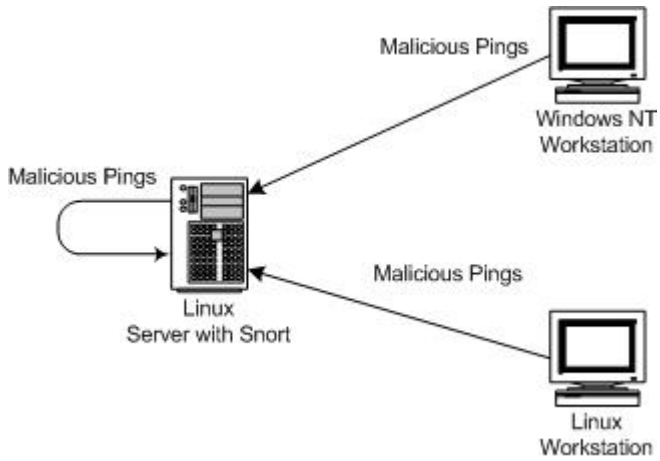


Figure 8: Testing Mechanism

Attack used:

Malicious Ping:

ping –s 1 –c *k ipaddress*

Snort sample command:

snort -l log –h 192.168.1.0/24 –c snort.conf –Y 10

-l option is for indicating the name of the log file that follows it. Here the log filename is given as “log”.

-h option is to specify the network that should be monitored. Here it is 192.168.1.0/24.

-c option lets the user give the name for the rule file that should be used by Snort. Here it is given as snort.conf.

-Y option lets the user specify the number of threads that will take care of the outputs for the generated alerts.

Note that standard Snort program will generate an error for the above command since –Y option is not available and was introduced by this project to indicate number of threads.

Testing Code:

For testing, the time after every 25 alerts was written into the status file. Functions that were written for that were:

void gettingtime(char *);

- to get the current time from the system.

float diffing(char *, char *);

- to compare the time after every 25 seconds.

Testing program generates alerts after run with following arguments:

```
Usage: a.out number_of_forks number_of_pings
```

The testing program is run on the Linux server which sends ping requests to, and accepts replies from, the Linux Workstation and Windows NT workstation.

The code for the Test program appears in Appendix B.

Some tests were also conducted running the test program on the Linux workstation to analyze results in an environment where Snort host is not under heavy load.

Sample Alert messages to be generated:

[**] [1:499:1] MISC Large ICMP Packet [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
12/01-05:24:35.560887 192.168.1.102 -> 192.168.1.101
ICMP TTL:32 TOS:0x0 ID:11812 IpLen:20 DgmLen:29
Type:8 Code:0 ID:1 Seq:2451 ECHO
[Xref => http://www.whitehats.com/info/IDS246]

[**] [1:499:1] MISC Large ICMP Packet [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
12/01-05:24:35.560887 192.168.1.101 -> 192.168.1.102
ICMP TTL:255 TOS:0x0 ID:6004 IpLen:20 DgmLen:29

Type:0 Code:0 ID:1 Seq:2451 ECHO REPLY
[Xref => http://www.whitehats.com/info/IDS246]

The above messages are generated according to the rule in Snort rule file for malicious pings. There are two messages generated as the first one is for ECHO packet for ping, and the other is for ECHO REPLY packet for ping.

The rule files associate a code for each type of errors, that is 1:499:1 here.

“MISC Large ICMP Packet” is the message for this type of attack.

Snort has a classification file that classifies all attacks. Here it is “Potentially Bad Traffic”.

Rule file associates Priority 2 with this attack.

The message has the date and time, and then destination and source information for this attack.

Then the message includes some basic packet header information for which alert is being generated.

At the end the message includes a link for a web page that has more information about this attack.

Test Hardware Configuration:

Testing environment consisted of a three-computer network networked by a hub. This was setup at the SAIT, Security and Assurance in Information Technology Lab at the Computer Science Department, Florida State University. Following is the hardware configuration for each of these machines.

Linux Server (Snort Host)	
Operating System	RedHat Linux 7.2
Processor	Pentium – III 800 MHz
RAM	256 MB
Swap Memory	512 MB
Network Card	3COM 3C905 Fast Ethernet 10/100Mbps

Linux Workstation	
Operating System	RedHat Linux 7.2
Processor	Pentium – MMX 200 MHz
RAM	32 MB
Swap Memory	64 MB
Network Card	3COM 3C905 Fast Ethernet 10/100Mbps

Windows NT Workstation	
Operating System	Windows NT 4.0 Workstation
Processor	Pentium – MMX 200 MHz
RAM	32 MB
Swap Memory	64 MB
Network Card	3COM 3C905 Fast Ethernet 10/100Mbps

Results:

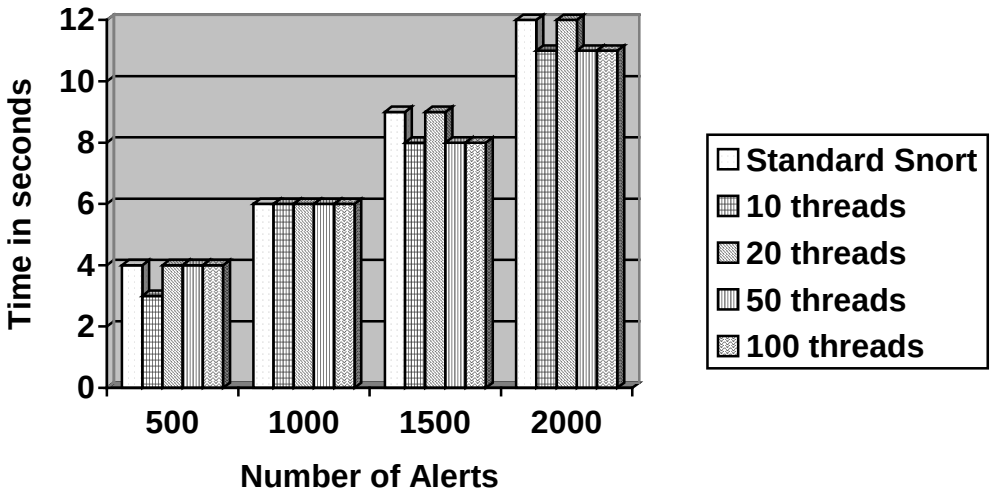
Threads have accelerated the overall process of alert outputs. The balancing factor here is the number of threads.

The test driver was first tested running on the same machine as Snort. That increased Snort host's overload for disk access for writing large number of alerts as well as taking care of the test program. The program was tested as follows:

Number of Attacking Processes: 100
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	4
10	500	3
20	500	4
50	500	4
100	500	4
Standard Snort (1 thread)	1000	6
10	1000	6
20	1000	6
50	1000	6
100	1000	6
Standard Snort (1 thread)	1500	9
10	1500	8
20	1500	9
50	1500	8
100	1500	8
Standard Snort (1 thread)	2000	12
10	2000	11
20	2000	12
50	2000	11
100	2000	11

Graph 1
a.out 100 10



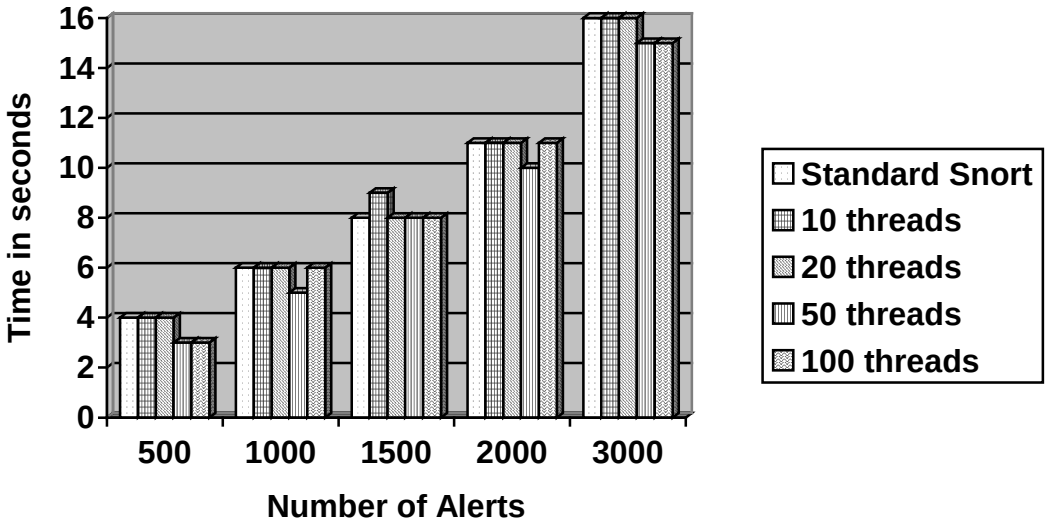
Observation:

For a lower number of attacks (500), a smaller number of threads is better as 10 threads perform faster as compared to the standard Snort. But there is not much difference between the Standard Snort program and a larger number of threads because of the overhead associated with creating more threads.

Number of Attacking Processes: 100
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	4
10	500	4
20	500	4
50	500	3
100	500	3
Standard Snort (1 thread)	1000	6
10	1000	6
20	1000	6
50	1000	5
100	1000	6
Standard Snort (1 thread)	1500	8
10	1500	9
20	1500	8
50	1500	8
100	1500	8
Standard Snort (1 thread)	2000	11
10	2000	11
20	2000	11
50	2000	10
100	2000	11
Standard Snort (1 thread)	3000	16
10	3000	16
20	3000	16
50	3000	15
100	3000	15

Graph 2
a.out 100 50



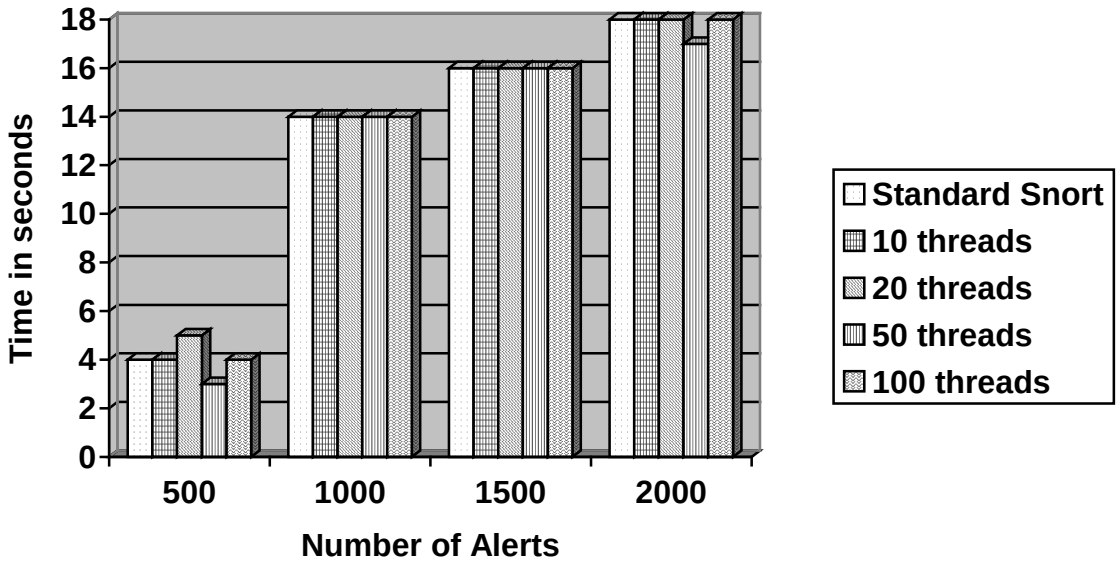
Observation:

Here the number of attacks per processes was increased. Since that increased the burden on the number of outputs, Snort with more threads took less time as compared to standard Snort. Even Snort with 10 or 20 threads was not enough to take care of the problem and acted almost only as good as the standard Snort. Snort with more threads did not show considerable difference in the middle because of the burden on the machine but with time, it started showing considerable difference.

Number of Attacking Processes: 250
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	4
10	500	4
20	500	5
50	500	3
100	500	4
Standard Snort (1 thread)	1000	14
10	1000	14
20	1000	14
50	1000	14
100	1000	14
Standard Snort (1 thread)	1500	16
10	1500	16
20	1500	16
50	1500	16
100	1500	16
Standard Snort (1 thread)	2000	18
10	2000	18
20	2000	18
50	2000	17
100	2000	18

Graph 3
a.out 250 10



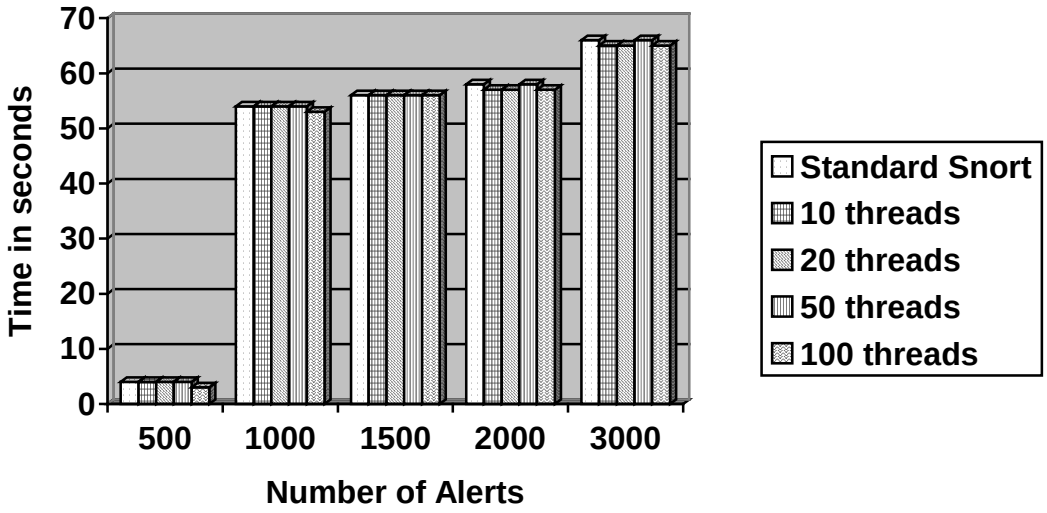
Observation:

With more forks for attacks (250 here), the load on the machine increased. Though it was for a lesser time since each process is making only 10 attacks. There is abrupt behavior in the beginning, then it all leveled. With time, Snort with 50 threads was better than Snort with fewer threads. It is even better than the Snort with 100 threads because Snort with 100 threads still was not able to balance load on the machine with the performance edge it offers.

Number of Attacking Processes: 250
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	4
10	500	4
20	500	4
50	500	4
100	500	3
Standard Snort (1 thread)	1000	54
10	1000	54
20	1000	54
50	1000	54
100	1000	53
Standard Snort (1 thread)	1500	56
10	1500	56
20	1500	56
50	1500	56
100	1500	56
Standard Snort (1 thread)	2000	58
10	2000	57
20	2000	57
50	2000	58
100	2000	57
Standard Snort (1 thread)	3000	66
10	3000	65
20	3000	65
50	3000	66
100	3000	65

Graph 4
a.out 250 50



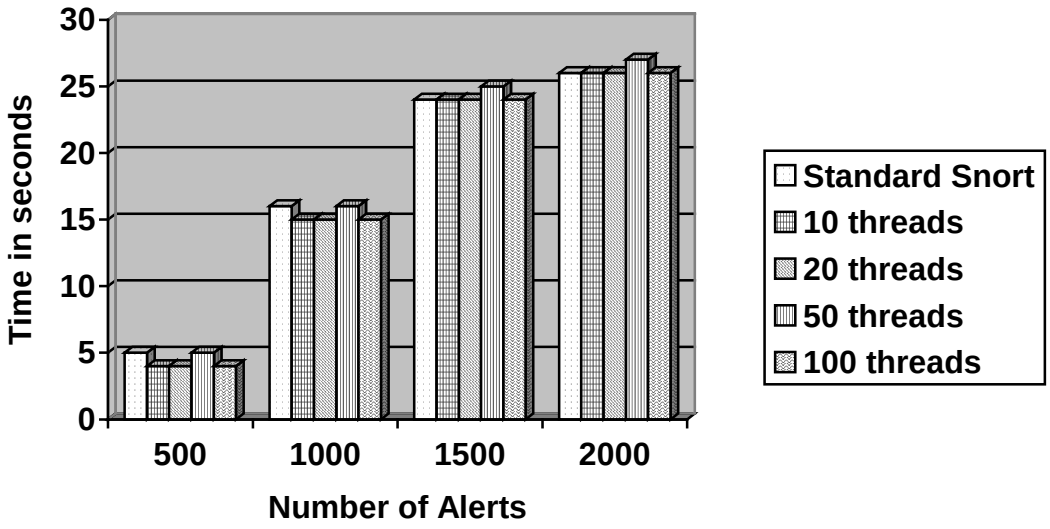
Observation:

With a larger number of attacks per process, the performance edge that more threads offer as compared to the standard Snort was visible for all number of threads as it was too much of a load to be handled by the standard Snort.

Number of Attacking Processes: 400
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	5
10	500	4
20	500	4
50	500	5
100	500	4
Standard Snort (1 thread)	1000	16
10	1000	15
20	1000	15
50	1000	16
100	1000	15
Standard Snort (1 thread)	1500	24
10	1500	24
20	1500	24
50	1500	25
100	1500	24
Standard Snort (1 thread)	2000	26
10	2000	26
20	2000	26
50	2000	27
100	2000	26

Graph 5
a.out 400 10



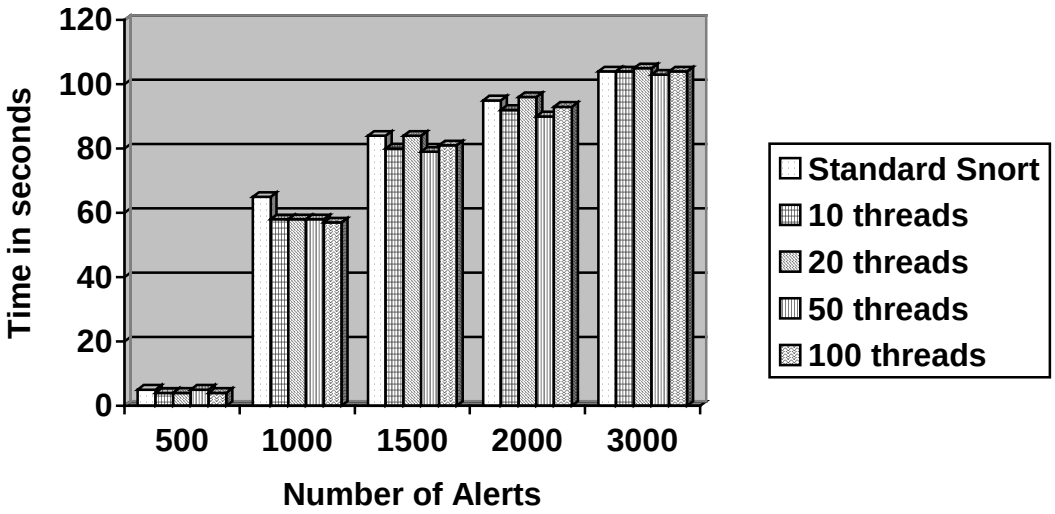
Observation:

In the beginning, Snort with more threads performed better but as the time progressed, 400 processes was too much of a load for the larger number of threads to handle. This shows that for a machine with increased load, a more powerful hardware configuration should be used.

Number of Attacking Processes: 400
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	5
10	500	4
20	500	4
50	500	5
100	500	4
Standard Snort (1 thread)	1000	65
10	1000	58
20	1000	58
50	1000	58
100	1000	57
Standard Snort (1 thread)	1500	84
10	1500	80
20	1500	84
50	1500	79
100	1500	81
Standard Snort (1 thread)	2000	95
10	2000	92
20	2000	96
50	2000	90
100	2000	93
Standard Snort (1 thread)	3000	104
10	3000	104
20	3000	105
50	3000	103
100	3000	104

Graph 6
a.out 400 50



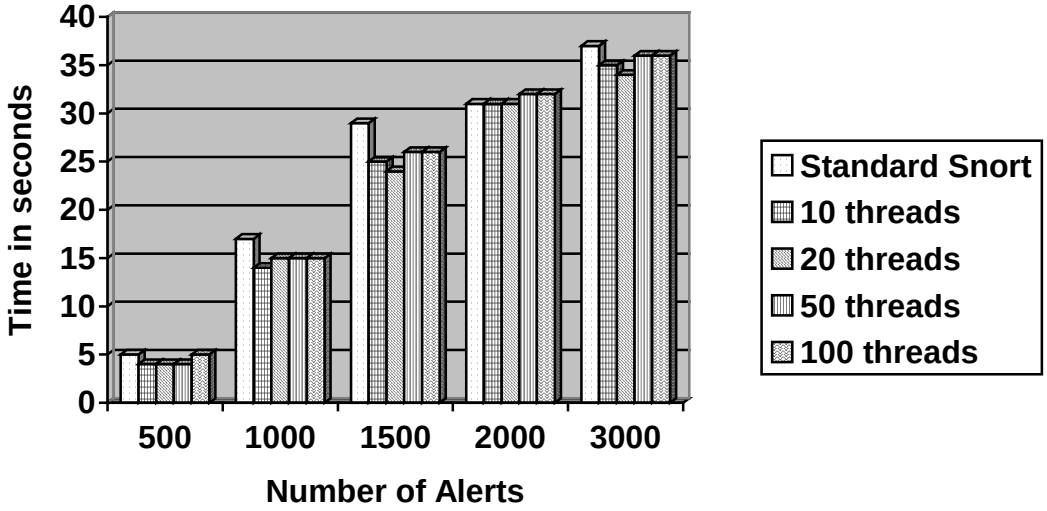
Observation:

This graph again shows the need to balance the number of attacks to the machine load in order to decide the number of threads. Though in the beginning, Snort with more threads performed better, with time Snort with 20 threads started to lag. So for the last comparison (3000 alerts), Snort with 50 threads performed the best.

Number of Attacking Processes: 500
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	5
10	500	4
20	500	4
50	500	4
100	500	5
Standard Snort (1 thread)	1000	17
10	1000	14
20	1000	15
50	1000	15
100	1000	15
Standard Snort (1 thread)	1500	29
10	1500	25
20	1500	24
50	1500	26
100	1500	26
Standard Snort (1 thread)	2000	31
10	2000	31
20	2000	31
50	2000	32
100	2000	32
Standard Snort (1 thread)	3000	37
10	3000	35
20	3000	34
50	3000	36
100	3000	36

Graph 7
a.out 500 10



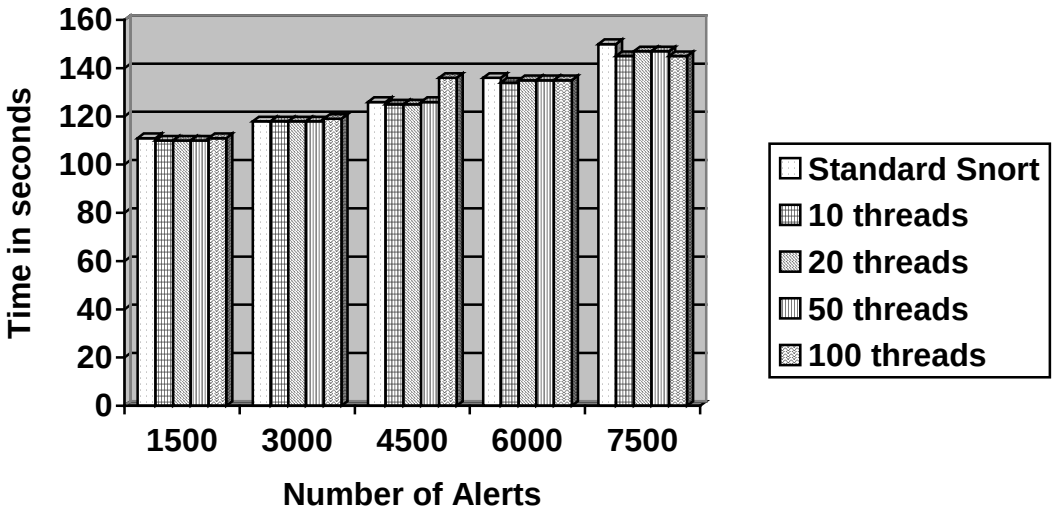
Observation:

Now we have 500 processes each making 10 attacks. Here we can see clearly the advantage of multi threaded Snort.

Number of Attacking Processes: 500
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	1500	111
10	1500	110
20	1500	110
50	1500	110
100	1500	111
Standard Snort (1 thread)	3000	118
10	3000	118
20	3000	118
50	3000	118
100	3000	119
Standard Snort (1 thread)	4500	126
10	4500	125
20	4500	125
50	4500	126
100	4500	126
Standard Snort (1 thread)	6000	136
10	6000	134
20	6000	135
50	6000	135
100	6000	135
Standard Snort (1 thread)	7500	150
10	7500	145
20	7500	147
50	7500	147
100	7500	145

Graph 8
a.out 500 50



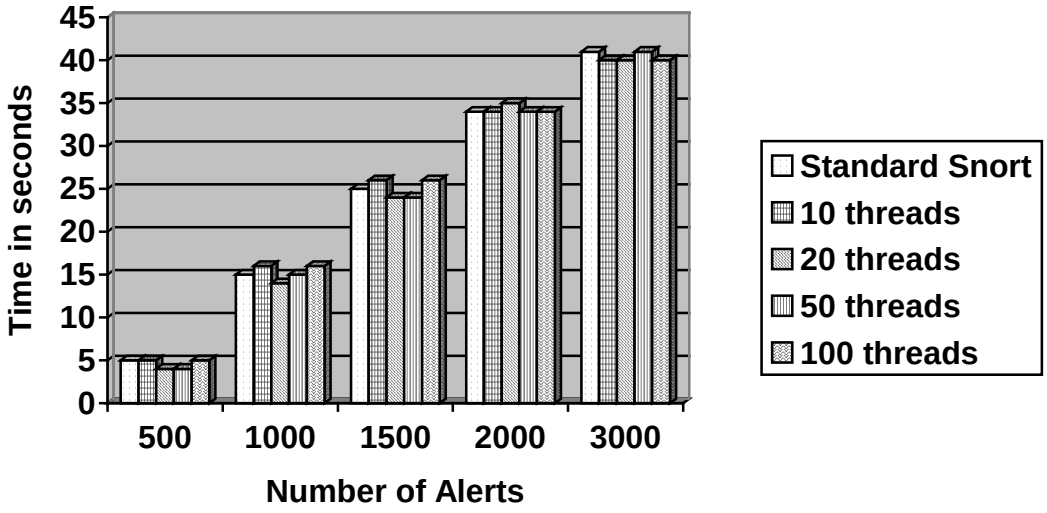
Observation:

With more attacks per process for 500 processes, the advantage begins to show. Here is where a more powerful machine would show better results for multi-threaded Snort.

Number of Attacking Processes: 600
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	5
10	500	5
20	500	4
50	500	4
100	500	5
Standard Snort (1 thread)	1000	15
10	1000	16
20	1000	14
50	1000	15
100	1000	16
Standard Snort (1 thread)	1500	25
10	1500	26
20	1500	24
50	1500	24
100	1500	26
Standard Snort (1 thread)	2000	34
10	2000	34
20	2000	35
50	2000	34
100	2000	34
Standard Snort (1 thread)	3000	41
10	3000	40
20	3000	40
50	3000	41
100	3000	40

Graph 9
a.out 600 10



Observation:

We have 600 processes attacking our network. We can clearly see the advantage of multi-threaded Snort.

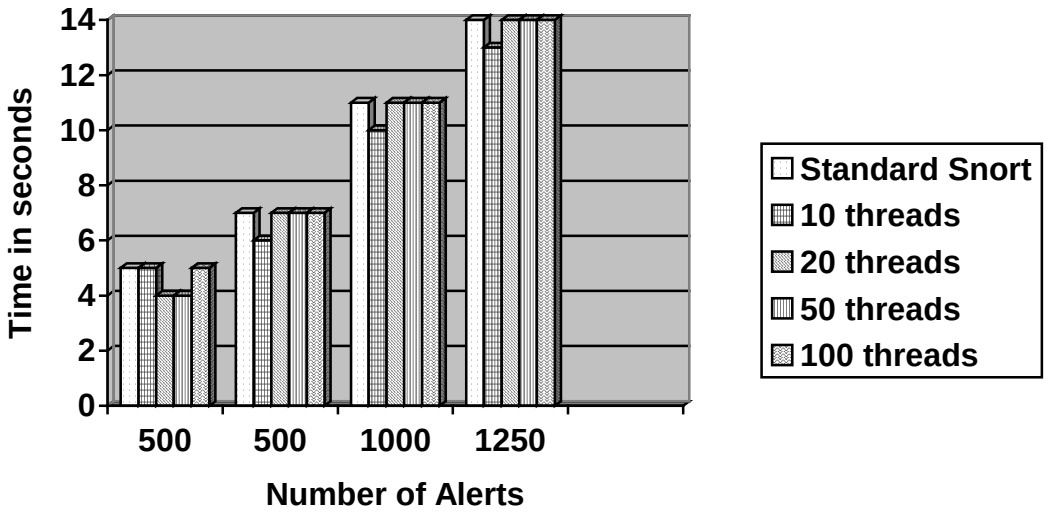
Next we tested our program by running the test program on the Linux workstation. What we expected was not any difference between Standard Snort and multi-threaded program since Snort machine is now relieved of the heavy load it was under for running the test program.

We tested for a maximum of 200 attacking processes for the machine limitation on our Linux workstation system.

Number of Attacking Processes: 100
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	7
10	500	6
20	500	7
50	500	7
100	500	7
Standard Snort (1 thread)	1000	11
10	1000	10
20	1000	11
50	1000	11
100	1000	11
Standard Snort (1 thread)	1250	14
10	1250	13
20	1250	14
50	1250	14
100	1250	14

Graph 10
a.out 100 10



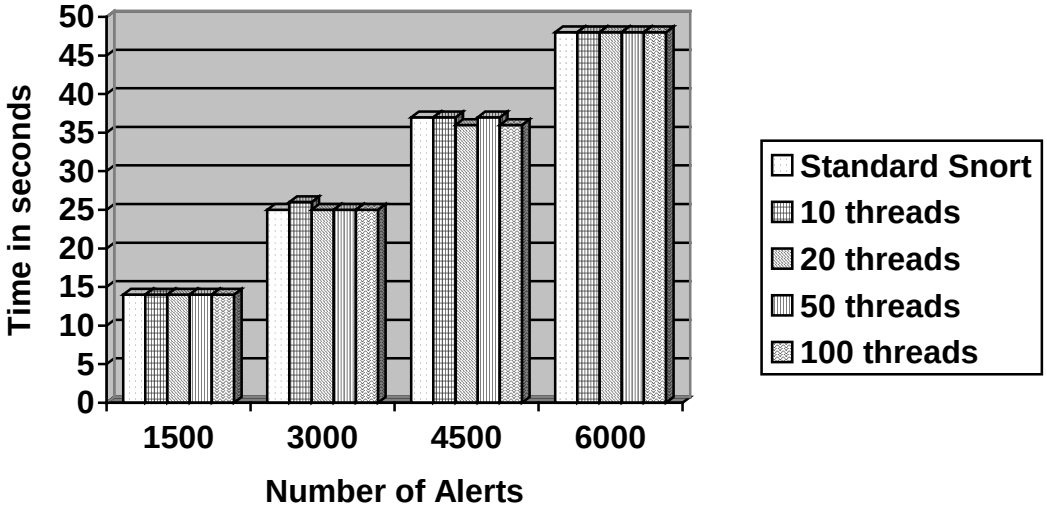
Observation:

There is some difference between the Standard Snort and multi-threaded snort up to a point where there are more context switches between threads and there the advantage the multi-threaded solution offers is balanced out.

Number of Attacking Processes: 100
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	1500	14
10	1500	14
20	1500	14
50	1500	14
100	1500	14
Standard Snort (1 thread)	3000	25
10	3000	26
20	3000	25
50	3000	25
100	3000	25
Standard Snort (1 thread)	4500	37
10	4500	37
20	4500	36
50	4500	37
100	4500	36
Standard Snort (1 thread)	6000	48
10	6000	48
20	6000	48
50	6000	48
100	6000	48

Graph 11
a.out 100 50



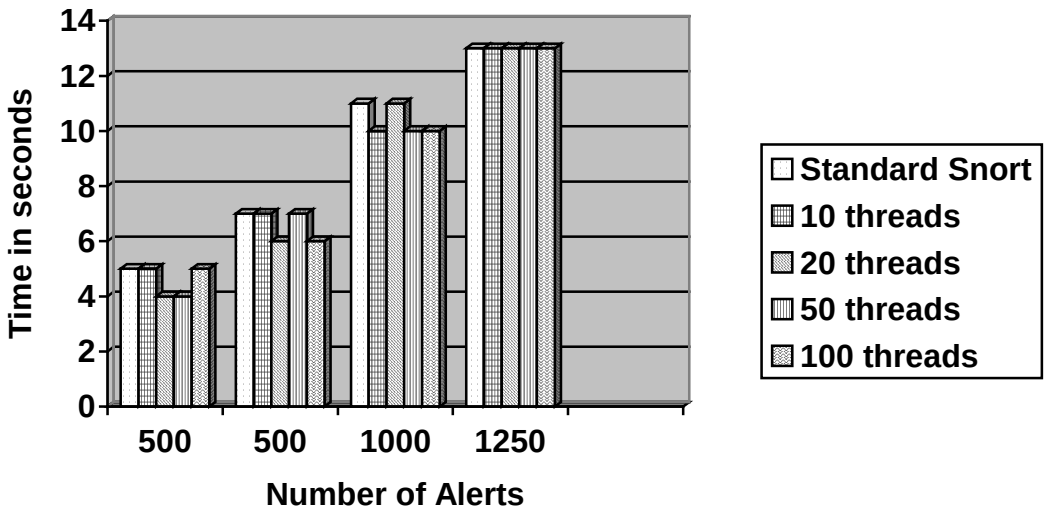
Observation:

There is almost no difference between the Standard Snort and Multi-threaded Snort. Though we see some difference in the middle, it all evens out as the number of threads switching increases. The advantage multi-threaded solution offers is not being availed here since there is not much load on the host machine.

Number of Attacking Processes: 200
Ping count (-c *k*): 10

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	500	7
10	500	7
20	500	7
50	500	6
100	500	7
Standard Snort (1 thread)	1000	11
10	1000	10
20	1000	11
50	1000	10
100	1000	10
Standard Snort (1 thread)	1250	13
10	1250	13
20	1250	13
50	1250	13
100	1250	13

Graph 12
a.out 200 10



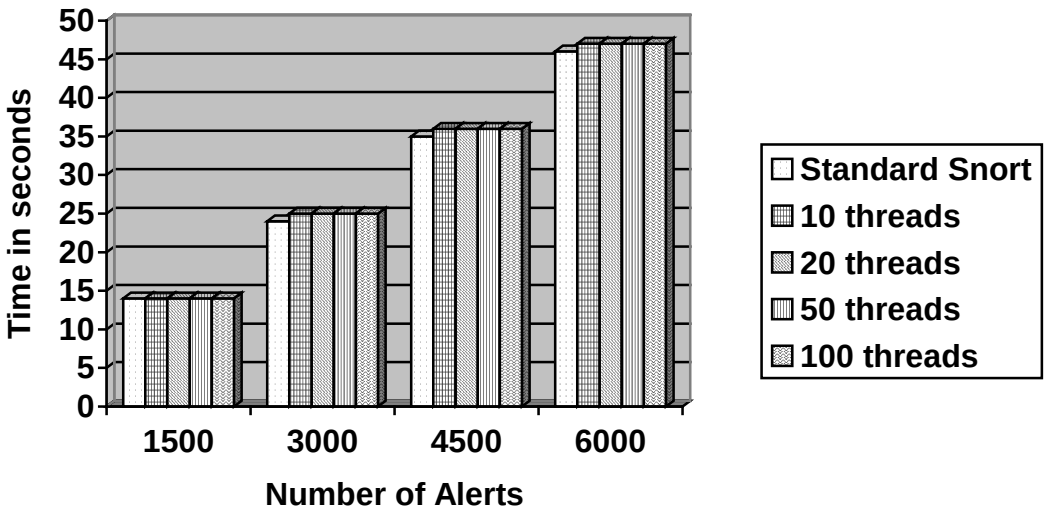
Observation:

Just like in 100 attacking processes, here we see that Standard Snort and multi-threaded Snort are performing equally well; the reason being the multi-threaded solution is not having a chance to show its usefulness in these test conditions.

Number of Attacking Processes: 200
Ping count (-c *k*): 50

Number of threads	Alert Landmark	Time taken (seconds)
Standard Snort (1 thread)	1500	14
10	1500	14
20	1500	14
50	1500	14
100	1500	14
Standard Snort (1 thread)	3000	24
10	3000	25
20	3000	25
50	3000	25
100	3000	25
Standard Snort (1 thread)	4500	35
10	4500	36
20	4500	36
50	4500	36
100	4500	36
Standard Snort (1 thread)	6000	46
10	6000	47
20	6000	47
50	6000	47
100	6000	47

Graph 13
a.out 200 10



Observation:

With more processes being involved in attack and with Snort host not having much load of its own, the Standard Snort performs better. Again, the advantage of multi-threaded Snort can be seen where the Snort host has a high load. An example would be the case where Snort alerts are being managed in a big database. Another would be Snort alets being sent to another machine.

Conclusion:

From the above tables for Snort host machine under heavy load, it is evident that even though a large number of threads is hampered by a lack of resources, they get the edge as the number of attacks increases. Our test environment writes output in a text file. So it should be noted that in an environment where output is directed to a more time consuming format, such as a database or windows pop up messages, threaded output should dramatically increase the performance. The test environment where Snort host was under a heavy load was where we could see the advantage multi-threaded Snort offers, but where Snort host was not under such load, there was not much advantage visible.

Another thing we see is that the number of threads should be carefully chosen depending upon the nature of environment, i.e. whether it is under more attacks from more locations or not.

Number of threads should be increased for an environment where attacks are in abundance (this requires a more powerful machine), or decreased to keep the number of threads low for an environment where fewer attacks.

In multi-threaded solution as the main program is relieved of the responsibility of handling the output in different ways, the process of analyzing the network data has become faster whenever there is higher disk latency and/or network delays are involved. In the case of repeated attacks from multiple sources, the multi-threaded Snort main process is not overloaded as threads are taking care of alert outputs. Though it should be noted that to process a larger number of threads, the machine that is hosting the new Snort should be powerful enough. The number of threads can be increased for an environment where attacks are in abundance, or decreased to keep number of threads low for an environment under less number of attacks.

In summary, we introduced multi-threaded output for Snort, and through our tests we found improved performance for environment where high disk latency and/or network delays are involved. We observed how variation in the number of threads is crucial to tune up multi-threaded Snort to run at its maximum potential. The Snort user, typically a Network administrator by changing the number of threads through command line argument, is able to adjust performance to suit his/her environment.

Future Work:

- Finding ways to add more Intelligence to Snort:
  - A way to improve the performance of Snort would be to introduce levels to parse the sniffed packets. As of now, Snort has a flat rule base and all rules need to be checked against the incoming packet. Introducing levels will only check against a set of rules depending upon the success or failure to match the earlier rule. That requires introducing new rule types. The design is to be changed so that the standard flow of Snort will be changed from being flat to being flexible to allow the direct jump into the depth of the rule base. The difficulties that lurk behind this idea are that to accomplish the different flow, the basic design of Snort needs to be changed. The challenge lies in keeping the intrusion detection capability of Snort along with introducing the more efficient parsing in levels. A detailed discussion is given in Appendix D.

- Incorporate Snort as a module in Linux kernel.
  - Kernel modules are faster and have a higher priority.

- Contribute in “Hank” Project: http://hank.sourceforge.net/docs/lwn.html#AEN80:
  - Hank is a new project modeled after Snort so it provides a new opportunity to build an Intrusion Detection system with improved design. New things can be applied and if there is an agreed (by the Hank and/or Snort community) design bottleneck for Snort, it can be avoided.

References:

A Silberschatz, J. Patterson, P. Galvin, Operating System Concepts, 3rd Edition, Addison-Wesley Publishing Company, 1991

Aslam, Taimur, Ivan Krsul and Eugene H. Spafford, "Use of A Taxonomy of Security Faults", Technical Report TR-96-051, Purdue University, Computer Science Dept., September 4, 1996

C. Warrender, S. Forrest, B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models", 1999 IEEE Symposium on Security and Privacy pp. 133-145

Carnegie Mellon Software Engineering Institute, CERT Coordination Center, 2000, "Writing rules and understanding alerts for Snort, a network intrusion detection system". Available.
http://www.cert.org/security-improvement/implementations/i042.14.html

Cisco Systems, 2002, "Q & A Cisco Secure Intrusion Detection System, General Concepts". Available.
http://www.cisco.com/en/US/products/sw/secursw/ps2113/products_qanda_item09186a008009d80d.shtml

Cisco Systems, 2002, "Cisco Intrusion Detection System, About this Guide",. Available.
http://www.cisco.com/en/US/products/sw/secursw/ps2113/products_user_guide_chapter09186a00800d9249.html

Denmac Systems Inc., Kevin Richards, 1999, "Network Based Intrusion Detection: A Review of Techbologies". Available.
http://www.denmac.com/images/intrusion_detection.pdf

Farmer, Daniel, and Eugene H. Spafford, "The COPS Security Checker System", Purdue University Technical Report CSD-TR-993, Purdue University, 1994

Internet Security Systems, Revision 1.68 - 7/9/2001, "RealSecure 6.0 Architecture, Technical Frequently Asked Questions". Available.
http://www.clico.pl/software/iss/RS6.0_Technical_FAQ_1.682.pdf

Internet Security Systems, 2002, "RealSecure ICEcap Manager Advanced Administration Guide", Version 3.5. Available.
http://www.isskk.co.jp/manuals/us/rsbli35/ICE_3.5_Adv_Admin.pdf

Internet Security Systems: "RealSecure 6.0 Architecture Technical Frequently Asked Questions", Rev 1.68, 7/9/2001

Internet Security Systems, 1998, White paper: “Network- vs. Host-based Intrusion Detection: A Guide to Intrusion Detection Technology”. Available. http://documents.iss.net/whitepapers/nvh_ids.pdf

Koral Ilgun, Richard Kemmerer, Philip Porras, State Transition Analysis: “A Rule based Intrusion Detection Approach”, IEEE Transactions on Software Engineering, 1995

Martin Roesch, “"Snort - Lightweight Intrusion Detection for Networks", USENIX LISA 1999 Conference

Martin Roesch, Chris Green, 2002, “Snort User Manual”. Available. http://www.snort.org/docs/SnortUsersManual.pdf

NFR Security, 2002, “NFR Security's Intrusion Management System – Overview”, Available. http://www.nfr.com/publications/white-papers/nfr-ims.pdf

Ohio University, Russ College of Engineering and Technology, School of Electric Engineering and Computer Science, Undated, “Comparison between Snort and NFR”. http://zen.ece.ohiou.edu/~nagendra/compids.html

Open Source Development Network, 2002, “Snort-users – 2002, Snort mailing list archive”. Available. http://www.geocrawler.com/archives/3/4890/2002/

Reto Baumann, Christian Plattner, 2002, “White Paper: Honeypots”. Available. http://security.rbaumann.net/papers.php

Sandeep Kumar, Eugene H. Spafford, "An Application of Pattern Matching in Intrusion Detection", Technical Report CSD-TR-94-013, The COAST Project, Purdue University, 1994

SecurityFocus, Karen Kent Frederick, 2001, “Network Monitoring for Intrusion Detection”. Available. http://online.securityfocus.com/infocus/1220

SourceForge, Todd Lewis, 2002, “Introducing Hank”. Available. http://hank.sourceforge.net/docs/lwn.html

W. Richard Stevens, Unix Network Programming Volume I, 2nd Edition, Prentice Hall, Inc., 1998

Appendix A: Sample Status file

Note: Attacks were also limited by the speed of the pings being generated.

Status file generated for:

100 threads	600 processes	10 pings per process

Number of attacks per seconds

```
For 25 took 1.000000
For 50 took 1.000000
For 75 took 1.000000
For 100 took 1.000000
For 125 took 2.000000
For 150 took 2.000000
For 175 took 2.000000
For 200 took 2.000000
For 225 took 2.000000
For 250 took 2.000000
For 275 took 3.000000
For 300 took 3.000000
For 325 took 3.000000
For 350 took 3.000000
For 375 took 3.000000
For 400 took 4.000000
For 425 took 4.000000
For 450 took 4.000000
For 475 took 4.000000
For 500 took 5.000000
For 525 took 5.000000
For 550 took 5.000000
For 575 took 6.000000
For 600 took 7.000000
For 625 took 8.000000
For 650 took 10.000000
For 675 took 11.000000
For 700 took 11.000000
For 725 took 12.000000
For 750 took 12.000000
For 775 took 13.000000
For 800 took 13.000000
For 825 took 13.000000
For 850 took 14.000000
For 875 took 14.000000
For 900 took 15.000000
For 925 took 15.000000
For 950 took 15.000000
For 975 took 16.000000
For 1000 took 16.000000
For 1025 took 17.000000
For 1050 took 17.000000
For 1075 took 18.000000
For 1100 took 18.000000
```

```
For 1125 took 19.000000
For 1150 took 19.000000
For 1175 took 20.000000
For 1200 took 21.000000
For 1225 took 21.000000
For 1250 took 22.000000
For 1275 took 22.000000
For 1300 took 23.000000
For 1325 took 23.000000
For 1350 took 24.000000
For 1375 took 24.000000
For 1400 took 24.000000
For 1425 took 25.000000
For 1450 took 25.000000
For 1475 took 25.000000
For 1500 took 26.000000
For 1525 took 26.000000
For 1550 took 27.000000
For 1575 took 27.000000
For 1600 took 28.000000
For 1625 took 28.000000
For 1650 took 29.000000
For 1675 took 29.000000
For 1700 took 29.000000
For 1725 took 30.000000
For 1750 took 30.000000
For 1775 took 31.000000
For 1800 took 31.000000
For 1825 took 31.000000
For 1850 took 32.000000
For 1875 took 32.000000
For 1900 took 32.000000
For 1925 took 33.000000
For 1950 took 33.000000
For 1975 took 34.000000
For 2000 took 34.000000
For 2025 took 34.000000
For 2050 took 35.000000
For 2075 took 35.000000
For 2100 took 36.000000
For 2125 took 36.000000
For 2150 took 37.000000
For 2175 took 37.000000
For 2200 took 37.000000
For 2225 took 37.000000
For 2250 took 37.000000
For 2275 took 38.000000
For 2300 took 38.000000
For 2325 took 38.000000
For 2350 took 38.000000
For 2375 took 38.000000
For 2400 took 38.000000
For 2425 took 38.000000
For 2450 took 38.000000
For 2475 took 38.000000
For 2500 took 39.000000
For 2525 took 39.000000
```

```
For 2550 took 39.000000
For 2575 took 39.000000
For 2600 took 39.000000
For 2625 took 39.000000
For 2650 took 39.000000
For 2675 took 39.000000
For 2700 took 39.000000
For 2725 took 39.000000
For 2750 took 40.000000
For 2775 took 40.000000
For 2800 took 40.000000
For 2825 took 40.000000
For 2850 took 40.000000
For 2875 took 40.000000
For 2900 took 40.000000
For 2925 took 40.000000
For 2950 took 40.000000
For 2975 took 40.000000
For 3000 took 40.000000
For 3025 took 41.000000
For 3050 took 41.000000
For 3075 took 41.000000
For 3100 took 41.000000
For 3125 took 41.000000
For 3150 took 41.000000
For 3175 took 41.000000
For 3200 took 41.000000
For 3225 took 41.000000
For 3250 took 41.000000
For 3275 took 41.000000
For 3300 took 41.000000
For 3325 took 42.000000
For 3350 took 42.000000
For 3375 took 42.000000
For 3400 took 42.000000
For 3425 took 42.000000
For 3450 took 43.000000
For 3475 took 43.000000
For 3500 took 43.000000
For 3525 took 43.000000
For 3550 took 44.000000
For 3575 took 44.000000
For 3600 took 44.000000
For 3625 took 45.000000
For 3650 took 45.000000
For 3675 took 46.000000
```

Appendix B: Testing Program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  int myrank, numforks, i, rank;
  char *cmdstr1, *cmdstr2, *cmdstr3;

  if (argc < 3) {
    printf("Usage: a.out number_of_forks(atleast 3)
number_of_pings\n");
    exit(0);
  }

  cmdstr1 = malloc(30);
  cmdstr2 = malloc(30);
  cmdstr3 = malloc(30);

  myrank = 0;
  rank = 1;
  numforks = atoi(argv[1]);
  sprintf(cmdstr1, "ping -s 1 -c %d 192.168.1.102", atoi(argv[2]) );
  sprintf(cmdstr2, "ping -s 1 -c %d 192.168.1.102", atoi(argv[2]) );
  sprintf(cmdstr3, "ping -s 1 -c %d 192.168.1.103", atoi(argv[2]) );

  for (i=0; i<numforks; i++) {
    if (!myrank) {
      if ( fork()==0) {
      myrank = rank;
      //    printf("I am %d\n", myrank);
      }
      else
      rank++;
    }
  }

  printf("I am %d\n", myrank);

  if ( (myrank%3) == 0) {
    printf("%s\n", cmdstr1);
    system(cmdstr1);
  }

  else if ( (myrank%3) == 1) {
    for(i=0; i<10000; i++);
    //    sleep(1);
    printf("%s\n", cmdstr2);
    system(cmdstr2);
  }

  else if ( (myrank%3) == 2) {
```

```
    for(i=0; i<10500; i++);
    // sleep(2);
    printf("%s\n", cmdstr3);
    system(cmdstr3);
  }

  else
    printf("what?!!\n");

  if(!myrank) {
    for (i=0; i<numforks; i++)
        wait(NULL);
  }
  return 0;

}
```

Appendix C: Project Code

I- multop.h (new file)

```
#include "snort.h"
#include <pthread.h>
#ifndef _MULTOP_
#define _MULTOP_

#define NO_T 100

int num_of_threads;

/* introducing threaded output data structure */

struct argsthread {
  Packet * p;
  OptTreeNode * otn;
  Event *event;
};

pthread_t outputthread[NO_T];
pthread_attr_t tattr[NO_T];
pthread_mutexattr_t mxattr[NO_T];
pthread_mutexattr_t mx2attr[NO_T];
pthread_mutex_t mx[NO_T];
pthread_mutex_t mx2[NO_T];
pthread_mutex_t itsme;
pthread_mutexattr_t itsmeattr;
struct argsthread *args1[NO_T];

pthread_condattr_t condatts[NO_T];
pthread_cond_t condts[NO_T];
int condints[NO_T];

pthread_cond_t locksig;

char *teststarttime;
char *testendtime;
float testdiff;
int numattacks;

struct thrarg {
  int myrank;
};

int mtcntr;
int sender;

FILE *fdmtcheck;

/* functions declarations */

void* outthreadfunc(void *);
```

```
void lockagain(int);
void mtinit(void);
void gettingtime(char *);
float diffing(char *, char *);


#endif /* for _MULTOP_ */
```

II- Changes in snort.c:

```
At the very top:
#include "multop.h"

Just after ParseCmdLine, call mtinit (at Line 145):
mtinit ();

Note: All the following functions can be added anywhere as far as these
functions are not inside any other function code. A good idea, that was
followed, is to add these at the very bottom of the file, where the
original code ends.
```

```
mtinit function is responsible for initializing all data structure and
creating threads initially)
```

```
void mtinit(void)
{
  //  struct thrarg *thrarg1[NO_T];
  struct thrarg *thrarg1[num_of_threads];
  int jj;

  mtcntr=0;
  sender=-1;

  //if ( (thrarg1 = malloc(sizeof(struct thrarg)))==NULL) {
  //  perror("malloc");
  //  exit(-1);
  // }
  pthread_mutexattr_init(&itsmeattr);
  pthread_mutexattr_setpshared(&itsmeattr, PTHREAD_PROCESS_SHARED);
  pthread_mutex_init(&itsme, &itsmeattr);
  printf("****************************************\n");
  printf("*********** num_of_threads is %d ********\n",
num_of_threads);
  printf("****************************************\n");

  teststarttime = malloc(50);
  testendtime = malloc(50);
  numattacks = 0;

  //  for (jj=0; jj<NO_T; jj++) {
  for (jj=0; jj<num_of_threads; jj++) {
    if ( (thrarg1[jj] = malloc(sizeof(struct thrarg)))==NULL) {
      perror("malloc");
      exit(-1);
    }
    thrarg1[jj]->myrank = jj;

    if ( (args1[jj] = malloc(sizeof(args1[jj])))==NULL) {
      perror("malloc");
      exit(-1);
    }
    if ( (args1[jj]->p = (Packet *)malloc(sizeof(Packet)))==NULL) {
```

```
      perror("malloc");
      exit(-1);
    }
    if ( (args1[jj]->otn = (OptTreeNode
*)malloc(sizeof(OptTreeNode)))==NULL) {
      perror("malloc");
      exit(-1);
    }
    if ( (args1[jj]->event = (Event *)malloc(sizeof(Event)))==NULL) {
      perror("malloc");
      exit(-1);
    }
    //    args1[jj] = malloc(sizeof(struct argsthread));
    pthread_mutexattr_init(&mxattr[jj]);
    pthread_mutexattr_setpshared(&mxattr[jj], PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&mx[jj], &mxattr[jj]);

    pthread_mutexattr_init(&mx2attr[jj]);
    pthread_mutexattr_setpshared(&mx2attr[jj], PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&mx[jj], &mx2attr[jj]);
    /*if ( pthread_mutex_lock(&mx[jj]) != 0) {
     perror("pthread_mutex_lock");
     exit(-1);
     }*/


    if (pthread_attr_init(&tattr[jj])!=0) {
      perror("pthread_attr_init");
      exit(-1);
    }

    if (pthread_create(&outputthread[jj], &tattr[jj], outthreadfunc,
(void  *)thrarg1[jj])!=0) {
      perror("pthread_create");
      exit(-1);
    }

    condints[jj] = 0;
    pthread_condattr_init(&condatts[jj]);
    pthread_condattr_setpshared(&condatts[jj], PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&condts[jj], &condatts[jj]);

    printf("thread %d created and mutex locked\n", jj);
  }
}
```

SIGUSR2 signal handler

```
void lockagain(int sig)
{
  //  if (pthread_mutex_lock(&mx[sender])!=0) {
  //  perror("pthread_mutex_lock");
  //  exit(-1);
  //}
  printf("trying to unlock %d\n", sender);
  pthread_mutex_trylock(&mx[sender]);
```

```
  printf("unlocked %d\n", sender);
  sender = -1;
}
```

time handling functions

```
void gettingtime(char *tsttime)
{
  time_t t;
  t=time(NULL);
  sprintf(tsttime, "%s", ctime(&t));
}

float diffing(char *firsttime, char *secondtime)
{
  char *rwday, *rmon, *rmday, *rhh, *rmm, *rss, *ry;
  char *lwday, *lmon, *lmday, *lhh, *lmm, *lss, *ly;
  char del[] = " :";

  struct tm rmt, lcl;
  time_t trmt, tlcl;
  float tdiffer;


  /********************** parsing ************************/

  rwday = strtok(firsttime, del);
  rmon = strtok(NULL, del);
  rmday = strtok(NULL, del);
  rhh = strtok(NULL, del);
  rmm = strtok(NULL, del);
  rss = strtok(NULL, del);
  /*rz = strtok(NULL, del);  */
  ry = strtok(NULL, del);

  lwday = strtok(secondtime, del);
  lmon = strtok(NULL, del);
  lmday = strtok(NULL, del);
  lhh = strtok(NULL, del);
  lmm = strtok(NULL, del);
  lss = strtok(NULL, del);
  /*lz = strtok(NULL, del);  */
  ly = strtok(NULL, del);

   /**************** parsing ends **********************/


   /*************** making time format ********************/

   /** for first **/

   rmt.tm_year = atoi(ry) - 1900;
   rmt.tm_mon = 7 - atoi(rmon);
   rmt.tm_mday = atoi(rmday);
   rmt.tm_hour = atoi(rhh);
   rmt.tm_min = atoi(rmm);
```

```
   rmt.tm_sec = atoi(rss);
   rmt.tm_isdst = -1;

   trmt = mktime(&rmt);

   /** for second **/
   lcl.tm_year = atoi(ly) - 1900;
   lcl.tm_mon = 7 - atoi(lmon);
   lcl.tm_mday = atoi(lmday);
   lcl.tm_hour = atoi(lhh);
   lcl.tm_min = atoi(lmm);
   lcl.tm_sec = atoi(lss);
   lcl.tm_isdst = -1;

   tlcl = mktime(&lcl);

   /********************* mk ends ************************/

   /********************* calculation *********************/

   tdiffer = difftime(tlcl, trmt);


   /***************** calculations end ******************/

   // if (tdiffer < 30) return 0;
  return (tdiffer);
}
```

III- Changes in rules.c:

Note: All the functions can be added anywhere as far as these functions are not inside any other function code. A good idea is to add these at the very bottom of the file, where the original code ends.

Original AlertAction function is totally replaced by the function below. Earlier it was used to generate alerts itself. Now it just signals a free thread to start work on the alert.

```
int AlertAction(Packet * p, OptTreeNode * otn, Event *event)
{

#ifdef DEBUG
    printf("        <!!> Generating alert! \"%s\"\n", otn->message);
#endif

    /* copy the Packet *p OptTreeNode * otn, Event *event */
    /* changes start */

    memcpy(args1[mtcntr]->p, p, sizeof(Packet));
    memcpy(args1[mtcntr]->otn, otn, sizeof(OptTreeNode));
    memcpy(args1[mtcntr]->event, event, sizeof(Event));


    //printf("I call\n");
    //    pthread_create(&outputthread, NULL, outthreadfunc, (void
*)args1);
    // pthread_mutex_unlock(&mx[mtcntr]);

    while (pthread_mutex_trylock(&mx[mtcntr])!=0) {
      mtcntr++;
      //      if(mtcntr == NO_T)
      if(mtcntr == num_of_threads)
      mtcntr=0;
    }
    //printf("got mtcntr as %d\n", mtcntr);
    pthread_mutex_unlock(&mx[mtcntr]);
    //    done = 0;
    // while(!done) {
      pthread_mutex_lock(&mx2[mtcntr]);
      //printf("i locked mx2[%d]\n", mtcntr);
      // if (condints[mtcntr]==0) {
      condints[mtcntr]++;
      pthread_cond_signal(&condts[mtcntr]);
      //    done=1;
      // }

      pthread_mutex_unlock(&mx2[mtcntr]);
      //}
      mtcntr++;
      //     if(mtcntr == NO_T)
      if(mtcntr == num_of_threads)
      mtcntr=0;
```

```
      //printf("I continue after signalling %d\n", mtcntr);

 /*#endif*/

    /* changes end */

    return 1;
}
```

outthreadfunc function is the function that runs when a thread is corrected. It assigns a rank to calling thread and waits on sigcondt.

```
void* outthreadfunc(void *args)
{

  int myrank;

  Packet * p; OptTreeNode * otn; Event *event;
  struct thrarg *args2;
  char *tempsttime;

  //  args2 = malloc(sizeof(struct thrarg));

  tempsttime = malloc(50);

  args2 = (struct thrarg *)args;
  myrank = args2->myrank;
  //printf("I am %d\n", myrank);

  p = (Packet *)malloc(sizeof(Packet));
  otn = (OptTreeNode *)malloc(sizeof(OptTreeNode));
  event = (Event *)malloc(sizeof(Event));

  sleep(3);

  while(1) {
    pthread_mutex_lock(&mx2[myrank]);
    //printf("thread called %d waits\n", myrank);
    //    while(condints[myrank]==0)
      pthread_cond_wait(&condts[myrank], &mx2[myrank]);
    condints[myrank]--;

    //printf("thread called %d got it\n", myrank);

    /* logging start time */

    if (numattacks == 0) {
      memset(testendtime, 0, sizeof(char)*50);
      gettingtime(teststarttime);
    }

    //if(startnow) {
    pthread_mutex_lock(&mx[myrank]);
```

```
    memcpy(p, args1[myrank]->p, sizeof(Packet));
    memcpy(otn, args1[myrank]->otn, sizeof(OptTreeNode));
    memcpy(event, args1[myrank]->event, sizeof(Event));

    /*
    p = args1[myrank]->p;
    otn = args1[myrank]->otn;
    event = args1[myrank]->event;
    */
    /* here is where I might let thread take care of CallAlertFuncs */
    CallAlertFuncs(p, otn->message, otn->rtn->listhead, event);

#ifdef DEBUG
    printf("   => Finishing alert packet!\n");
#endif

    if(p->ssnptr != NULL)
      {
      if(AlertFlushStream(p, p->ssnptr) == 0)
        {
          CallLogFuncs(p, otn->message, otn->rtn->listhead, event);
        }
      }
    else
      {
        CallLogFuncs(p, otn->message, otn->rtn->listhead, event);
      }

    pthread_mutex_lock(&itsme);


    /* logging end time and comparing */
    numattacks++;

    if( (numattacks % 25) == 0) {
      memset(testendtime, 0, sizeof(char)*50);
      memset(tempsttime, 0, sizeof(char)*50);
      gettingtime(testendtime);
      sprintf(tempsttime, "%s", teststarttime);
      testdiff = diffing(tempsttime, testendtime);
    }

    /* writing into status file */
    if( (numattacks % 25) == 0) {
      fdmtcheck=fopen("/root/log/fdmtcheck.txt", "a");
      fprintf(fdmtcheck, "For %d took %f\n", numattacks, testdiff);
      fclose(fdmtcheck);
    }

    sender = myrank;
    pthread_mutex_unlock(&mx[myrank]);

    pthread_mutex_unlock(&mx2[myrank]);

    // earlier approach
    /*if(raise(SIGUSR2)!=0)
      printf("couldnt raise the signal");
```

```
      printf("raised by %d\n", myrank);*/
    //while(sender!=-1);
        pthread_mutex_unlock(&itsme);


    memset(args1[myrank]->p, 0, sizeof(Packet));
    memset(args1[myrank]->otn, 0, sizeof(OptTreeNode));
    memset(args1[myrank]->event, 0, sizeof(Event));
  }
  pthread_exit(NULL);
  printf("naheen hua\n");

#ifdef DEBUG
  printf("   => Alert packet finished, returning!\n");
#endif
  return(NULL);
}
```

Appendix D: Introducing Levels in Snort – Potential Project Overview

Performance Enhancement in Snort Rules Parsing by introducing levels

Identifying work area:

As of now, Snort has a flat rule base and all rules need to be checked against the incoming packet. Introducing levels will only check against a set of rules depending upon the success or failure to match the earlier rule. That requires introducing new rule types. The design is to be changed so that the standard flow of Snort will be changed from being flat to being flexible to allow the direct jump into the depth of the rule base. The difficulties that lurk behind this idea are that to accomplish the different flow, the basic design of Snort needs to be changed. The challenge lies in keeping the intrusion detection capability of Snort along with introducing the more efficient parsing in levels. A detailed discussion is added within his report.

Current Scenario:

Snort rule base architecture is a flat one, that is, all the rules must be checked for all the packets (except Dynamic ones, that will be discussed below). In an environment where attacks are less frequent, this approach results in unnecessarily waste of the resources.

Relation between Rules:
OR, AND, Activate-Dynamic

OR model is between all the rules. So if any one of the rules is matched for a packet, an alert is generated.

AND model is within the Rule. So that all the conditions within the rule must be satisfied to generate an alert.
Example:
alert tcp $EXTERNAL_NET any -> $HOME_NET 70 (msg:"MISC gopher proxy"; content: "ftp|3a|"; content: "@/";)

Above example shows the AND within a rule. An alert is generated if all of the following conditions are satisfied:

- Packet protocol is TCP.
- The source network is a network defined by the variable EXTERNAL_NET somewhere else in the rule file.
- Source port can by any.
- The destination network is a network defined by the variable HOME_NET somewhere else in the rule file.

- Destination port is 70.
- Content of the packet has the pattern "ftp|3a|".
- Content of the packet has the pattern "@/".

Observation:
There is no AND between different rules.

Activate – Dynamic Relation:
Activate rules are matched to the packets, and when there is a match, a corresponding Dynamic rule is activated. Dynamic rules are not checked for any packets until those are not activated by an Activate rule.
First matched Activate Rule and its corresponding Dynamic rules do not apply to the same packet.

What can be extended:

An AND model between rules should be introduced. That means to introduce Depth into parsing, so that following flow will be followed:

IF Rule1 is true
Check for Rule2
Else
Goto Rule3

This will prevent checking for all the rules in the rule-base.

Project Design:

Two new Action Types will have to be introduced:

Intell *k*

- For indicating it is an intelligent rule.
- *k* is an integer.
- Seeing this type of rule, Snort will match the conditions given within this rule, and if all are matched, Snort will find the next intell or intell_f rule with index *k* to be matched. Note that those other rules do not need to be matched unless first (in order) *k* intell rule is matched.

intell_f *k*

- For indicating the end of the intell depth
- This tells the Snort that this rule is the final of the *k* line of intell rules.
- If this rule is also matched, then an alert will be generated.

intell 1 tcp $EXTERNAL_NET any -> $HOME_NET 70 (msg:"MISC gopher proxy"; content: "ftp|3a|";)

intell_f 1 Alert tcp $EXTERNAL_NET any -> $HOME_NET 70 (msg:"MISC gopher proxy"; content: "@/";)

In the above example, index *k* is 1. So when the rules are being parsed, the relation between the two rules is identified. First rule is of intell type, which matches for the pattern "ftp|3a|". The second rule, which is of intell_f type, is used only when the first rule is satisfied. The second rule matches for the pattern "@/", and if that is matched, an alert is generated with the given message.

Places where changes are to be introduced:

Standard Snort has to be changed to introduce the new rule types intell and intell_f. While creating rtn (Rule Tree Node), the new Rule type has to be accommodated. Also the place where packets are matched with the rules needs to be modified.

So there are two different stages that need to be taken care of:

Stage A – Putting intell and intell_f in rtn
Stage B – Parsing the packets

Implementation:

New Data Structures:

#define RULE_INTELL 12

- Standard Snort uses 11 different rule types to be used as Rule types. So intell types can use 12 to indicate their Rule type.

struct _OutputFuncNode *IntellList;

- Intell function list
- _OutputFuncNode structure maintains the list of output functions. IntellList will be a new variable of this structure to keep track of intell functions.

ListHead Intell;

- Intell Block Header
- This will keep track of the Block headers for Intell rules.

Functions to be affected:

ParseRulesFile

- Reading the rule file

- This function takes care of way a rule file is read and parsed to get the rules from it. Standard Snort does it sequentially, but for this project, it needs to be done in a different order. Whenever an intell type rule is encountered, the next rule to be fetched is not the one following it, but the next intell rule of the same index needs to be found from the file.

ParseRule

- Processsing an individual Rule
- Since the format of intell and intell_f rules is different, they need to be parsed in a different way.

RuleType

- Determines what type of rule is being processed and returns its equivalent value
- With other rule types, this function needs to be able to recognize the intell and intell_f rule types.

CreateDefaultRules

- Creates a Rule Type in Rule Type Tree
- This function needs to be changed to be able to add intell and intell_f to the Rule Type Tree.

CreateRuleType

- Creates a new type of rule and adds it to the end of the rule list
- This function needs to be changed to be able to create intell and intell_f rule types and add those to the rule list.

ParsePacket

- When the packet is being parsed to be matched against the rules
- This function needs to be changed so that it can also be matched against the intell and intell_f type rules.
- Notice here that a packet might have to be checked many times if it keeps on satisfying intell rule in depth, but the process to match it against an intell index needs to be stopped whenever there is a mismatch and the following intell and intell_f rules for that index need to be ignored.

New functions to be written:

IntellParse:

- This function will parse through the Intell Tree.

IntellSeeNext:

This function will be responsible to get to next Intell / Intell_f.

int IntellAction:

This function will return the integer value for what Action to take finally.

ParseIntellMessage

This function will be responsible for the alert message to propagate.

Flow Change:

ParseRulesFile()

- ParseRulesFile has to run after RuleType to see if RuleType returned RULE_INT; to accommodate number of tokens for ParseRules
- This is a big deviation from Snort's architecture
- ParseRulesFile passes one line at a time as a rule to ParseRule. But in our case ParseRule should run until it gets to the bottom of the intelligent rules.

ParseRule()

- ParseRule runs once for each rule. In our case it needs to run multiple times. This is only possible by adding conditions in ParseRulesFiles() and providing ParseRule() with more than one line.
- But that means Parsing is overlapping between ParseRulesFile() and ParseRule()

CheckRule()

- CheckRule needs to be relaxed to accommodate our new Rule type.

Toks[x] array for tokenizing rules

- This array needs to be modified throughout the program to accommodate new number of tokens which will expand through the program.

ProcessHeadNode ()

- ProcessHeadNode needs to be accessed more than once.

Performance Comparison Expectation:

- Snort with levels should be a bit faster as compared to the Standard Snort since all the rules need not be checked.
- Snort with levels could be a bit slower if an Intell action rule is satisfied, to the degree of depth.

Focal Complexity:

Project flow needs to be changed. Since Snort was made without any vision of levels where AND would be provided between different rules, the existing flow does not accommodate that idea at all. Thus flow of the program needs to be changed, and that brings more complexities into the program by hindering what is the basic function of Snort. So the challenge is to introduce levels and keep Snort working.

The End