IMPROVING PERFORMANCE OF NETWORK INTRUSION DETECTION SYSTEMS THROUGH CONCURRENT MECHANISMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA ATAKAN

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

DECEMBER 2003

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Cevat Şener
Supervisor

Examining Committee Members

Asst. Prof. Dr. İbrahim Körpeoğlu _____

Dr. Atilla Özgit _____

Dr. Onur Tolga Şehitoğlu _____

Dr. Cevat Şener _____

Ms. Burak Dayıoğlu _____

ABSTRACT

IMPROVING PERFORMANCE OF NETWORK INTRUSION DETECTION SYSTEMS THROUGH CONCURRENT MECHANISMS

Atakan, Mustafa

MSc, Department of Computer Engineering

Supervisor: Dr. Cevat Şener

December 2003, 48 pages

As the bandwidth of present networks gets larger than the past, the demand of Network Intrusion Detection Systems (NIDS) that function in real time becomes the major requirement for high-speed networks. If these systems are not fast enough to process all network traffic passing, some malicious security violations may take role using this drawback. In order to make that kind of applications schedulable, some concurrency mechanism is introduced to the general flowchart of their algorithm. The principal aim is to fully utilize each resource of the platform and overlap the independent parts of the applications. In the sense of this context, a generic multi-threaded infrastructure is designed and proposed. The concurrency metrics of the new system is analyzed and compared with the original ones.

Keywords: NIDS, snort, concurrency, multi-threading, rule-based, real-time

ÖZ

AĞ SALDIRI TESPİT SİSTEMLERİNDE EŞ ZAMANLI PROGRAMLAMA TEKNİKLERİ KULLANARAK PERFORMANS ARTIRIMI

Atakan, Mustafa

Y. Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Dr. Cevat Şener

Aralık 2003, 48 sayfa

Bilgisayar ağlarında bant genişliği arttıkça, ağ üzerinden akan trafiği gerçek zamanlı şekilde işleyecek Ağ Saldırı Tespit Sistemleri (ASTS), günümüz bilişim teknolojisinin en büyük ihtiyaçlarından biri haline gelmiştir. Bu sistemlerin yüksek hızda akan trafiği işleyebilecek şekilde dizayn edilmemesi ciddi güvenlik açığı sorunlarına neden olabilir. Bundan dolayı, ASTS'lerin genel algoritma akışı içerisinde birtakım eş-zamanlı programlama teknikleri kullanılmalıdır. Amaç, ASTS'lerin çalıştığı donanım üzerindeki kaynakları en verimli şekilde kullanabilmesi ve birbirinden bağımsız program parçacıklarının eş zamanlı olarak çalışabilmesidir. Bu bağlamda çok-görevli yeni bir yapı tasarlanmakta ve eş zamanlı ölçütlerle incelenerek önceki yapılarla karşılaştırılmaktadır.

Anahtar Kelimeler: ASTS, snort, eş-zamanlılık, çok-görevli programlama, kural-tabanlı, gerçek-zamanlı

To ones who dedicate themselves for the well-being of humanity

ACKNOWLEDGMENTS

I thank

my parents, Gülümser and Osman Atakan, for their endless love and support,

my unique darling, Aslı Akar, for her love, support, sweetness and understanding,

my brothers Ali and Ayhan Atakan, sister-in-love Tuba Atakan, friend Özlem Uludüz for their motivating promotion,

my supervisor Dr. Cevat Şener, for his spiritual hand standing me up when I have fallen down,

Mr. Burak Dayıoğlu, for his guideness and informational help,

my job fellows Çağlar Bilir, Ahmet Öztürk, Gökhan Eryol, Fehmi Turan for their technical support, my friends Ali Güler, Burak Bilen, Feyza Taşkazan and Tayfun Asker for their understanding and help when I am busy with my thesis,

my friend Ersan Topaloğlu for his hints and clues,

ODTU Computer Center stuff for the technical infrastructure they supply,

and all other friends giving me help in producing this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Network Intrusion Detection Systems

Intrusion Detection System (IDS) is a kind of security management system for computers and networks. It is based on the assumption that an intruder can be detected through an analytical inspection and evaluation of various parameters such as network traffic, CPU utilization, I/O utilization, user location, and various file activities [1]. One of the famous types of IDSs is Network Intrusion Detection System (NIDS). It is an important part of any network security architecture. It constructs an outer defense shell which tries to match network traffic and predefined suspicious activity or patterns, and alert system administrators when potential hostile traffic is detected.

1.2 Rule Based Network Intrusion Detection Systems

As for the working mechanism, Rule Based Network Intrusion Detection System (RB-NIDS) is one of the approaches in NIDS. The philosophy behind is to use expert systems technology to analyze automatically network trail data for intrusion attempts

of pending or completed security violations. An expert system, where knowledge about a problem domain is represented by a set of rules, performs pattern matching techniques to determine a match between observed data and the rules. These predefined rules are actually abstract (high-level) patterns of malicious activity. They can be constructed anything from port numbers in the header to a specific byte sequence in the payload of the packet. The purpose of the technology is to seek to increase the availability of computer and network systems protected by detecting and eliminating the intrusions.

In rule-based intrusion detection systems, there are two approaches: preemptory and reactionary. The difference is all about time [2]:

- In the preemptory approach, your intrusion detection tool actually listens to network traffic. When a suspicious activity is detected(a flood of particular packets, for example), the system takes appropriate action.

- In the reactionary approach, your intrusion detection tool watches your logs instead. Again, when any suspicious activity is noted, the system takes appropriate action.

This distinction may seem so unimportant, but there is a major difference. The reactionary approach is simply one upper model of standard logging; it alerts you to the fact that an attack has just taken place, even if that means 3.5 seconds ago.

In contrast, the preemptory approach actually lets your system to be responsive at the time of an attacker is executing his intrusion statements. In order to take banning operations, live operators are armed to witness and track an attack in progress. In order to satisfy the preemptory property, some tuning techniques may be recommended. There are many algorithmic alternatives in the books and papers for this

goal. However, there is also a different perspective such as concurrency.

1.3 Concurrency Paradigm

Concurrent paradigm provides a way to organize software that contains relatively independent parts. It also provides a way to make use of multiple processors. There are three broad, overlapping classes of concurrent applications- multi-threaded systems, distributed systems, and parallel computations- and three corresponding kinds of concurrent programs [3].

The term multi-threaded usually means that a program contains more light-weight processes (threads) than there are processors to execute the threads. When one of the threads is blocked for a system call (e.g. making I/O request), preemption takes place and one of the ready threads is awakened to execute according to the scheduling policy. By doing so, the task of the thread making blocking system call and the task of the next thread to execute is overlapped, even in a single processor computer. The second broad class of applications is distributed computing, in which components execute on machines connected by a local or global communication network. Therefore, the processes communicate by exchanging messages. The components in a distributed system are often themselves multi-threaded programs. Parallel computing is the third broad class of concurrent applications. The goal is to solve a given problem faster or equivalently to solve a larger problem in the same amount of time. These kinds of computations are large and compute-intensive.

1.4 Performance Problem of RBNIDS

It is seen that preemptory approach in a RBNIDS has a major advantage over reactionary approach. Although , in reactionary approach (also known as offline), it is possible to examine the logs in great detail by shifting the processing of audit information to non-peak times, it can only detect intrusion after the fact. On the other hand, preemptory approach (also known as real time IDS) can potentially catch intrusion attempts before the system is compromised. However, preemptory model, like all intrusion detection systems, will negatively affect system performance due to their collecting and processing of audit trail information. For example, early prototyping of a real time RBNIDS on a UNIX workstation showed the algorithm was using up to 50% of the available processor throughput to process and analyze the audit trail [4]. Since preemptory model is resource-intensive, this approach represents several problems: One problem is due to inherent interactivity of such systems. If an attacker knows that you're running a preemptory intrusion detection system, he can make several assumptions. One is that your IDS will undertake an identical action when met with an identical attack. Hence, by flooding your host with multiple instances of the same attack from different addresses, he can perform a process saturation attack and perhaps crash or incapacitate your IDS. For example, what if your IDS invokes a shell to execute commands when under attack? How many shell processes does it take before your system will malfunction? The other problem is due to hardware and software limitations. What about the preemptory RBNIDS isn't as fast as the speed of the packets accumulating in the network buffer of the kernel? After the buffer is full, the IDS software begins to miss newly incoming packets where the actual compromising sequence is hidden.

1.5 Proposed Solutions

One of the solution is, depending on your processor power and memory limitations, you may be forced to choose traffic analysis over content analysis. Traffic analysis is less resource-intensive because you're processing packet headers and not content. This protects against many attacks, but not all. A substantial number of attack signatures are buried in packet content, and simple traffic analysis is insufficient for these. Hence, this alternative seems awkward.

Another solution, and also the focus of this paper, is full utilization of computer resources by introducing the concurrency paradigm to the application. By categorizing the independent parts of a RBNIDS, one can fairly deploy those parts to the resources in order to get maximum speedup.

1.5.1 Determining Concurrency Metrics

Considering the introduced problems of a preemptory RBNIDS, it can be assumed that the major requirement for such a system is ability to function in "real time". Real time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable. The mentioned functionality is in the latter category.

As a second assumption, the event that the packet is arriving to the network card can be taken as periodic (occurring in regular interval). Although this was not the case, the event could be simulated as periodic by taking an average on the network traffic. If it's so, a schedulable formula[1] can be devised for the processing application

[1] If there are m periodic events and event occurs with period P_i and requires T_i seconds of CPU

[5].

A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it my not even be possible to handle them all. As an example, consider a soft real time system. It has a single CPU and an 100 Mbps interface. Assume that average packet length transmitting over this interface is 1Kb. If the switch of the interface is capable of fully utilizing the channel, 100000 (100Mb/1Kb) packets arrive in a second. That means, period P of that event is 10 micro seconds (1/100000). The application is said to be schedulable only if the processing time of it is less than 10 microseconds. Implicit assumption in this calculation is that there is no other process apart from the one that is analyzed. If a preemptory RBNIDS is not schedulable, the processing time T must be reduced to a level that satisfies the above criterion. Suppose that the execution time of above example was 20 microseconds, meaning it is not schedulable. By programming the same application concurrently on a two-processors, the execution time could be reduced, say it, 10 microseconds. Then, the speedup[2] of the new schedulable application is 2. This is a linear speedup[3] and makes the another metric, the efficiency[4] of the application, 1.0. Both metrics depend on the number of processors, the problem size

---

time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \left(\frac{T_i}{P_i} \right) <= 1 \tag{1.1}$$

A real time system that meets this criterion is said to be schedulable.

[2] Let T be the execution time for solving some problem using a sequential program executed on a single processor. Let T_p be the execution time for solving the same problem by introducing some concurrent processes that is executed on p processors. Then the speedup of the application is defined as

$$Speedup = \frac{T}{T_p} \tag{1.2}$$

[3] If the speedup on p processors is p, then this is called linear (perfect) speedup.

[4] Efficiency is a dual measure of speedup. It can be expressed as:

$$Efficiency = \frac{Speedup}{p} \tag{1.3}$$

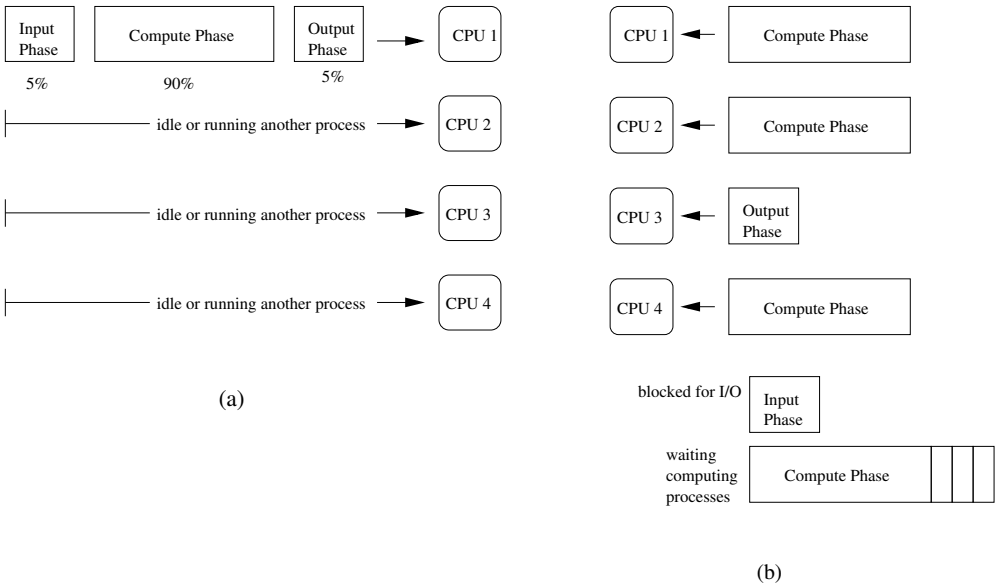

Figure 1.1: (a) Fundamental phases of a sequential RBNIDS, (b) Concurrent overview of these phases on a 4-processors machine

and the algorithm that is used.

1.5.2 Introducing Concurrency to a Preemptory RBNIDS

A typical RBNIDS, as in Fig. 1.1(a), has three phases. Input, compute(detection), and output. Suppose that in a sequential program, the input and output phases each take 5% of the execution time and the compute phase takes the remaining 90%. Although the input and output phases are inherently sequential (they can't be made faster by concurrent programming), they can be overlapped if each uses a different system resource- input from network card and output to disk and using different busses. But for the sake of simplicity, both of them are supposed to be non-overlapping. Then, the Amdahl's Law[5] says that the best speedup that this concurrent program can achieve

[5] A sequential program consists of an execution time interval that can not be parallelized $Time_{seq}$ and the remaining interval that can be parallized $Time_{par}$ with a partial speedup $Speedup_{part}$. Then the ovearall speedup formula can be expressed as:

$$Speedup_{overall} = \frac{1}{Time_{seq} + \left(\frac{Time_{par}}{Speedup_{part}}\right)} \tag{1.4}$$

is ten. No matter how many processors are used, the input and output phases will still remain 10%. The minimum time for any concurrent applications will be more than that percentage even if the compute phase is reduced to nearly zero seconds.

When applying concurrency techniques to a sequantial RBNIDS, the computation phase of the application can be multiplied as seen in Fig. 1.1(b). Since the detection computations of each packet are independent from each other, some of them can execute in different CPUs of the system concurrently. The input and the output threads can also execute when it is their turn in scheduling policy. They should have higher priority than the detection threads, because they usually block more often than the others. So when they are blocked for making I/O operations, for example, the detection threads can still continue to do their tasks.

1.6 What is Snort?

Snort is a libpcap-based [6] packet sniffer and logger that can be used as a lightweight network intrusion detection system (NIDS). It features rules based logging to perform content pattern matching and detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and much more. Snort has real time alerting capability, with alerts being sent to syslog, Server Message Block (SMB) "WinPopup" messages, or a separate "alert" file. Snort is configured using command line switches and optional Berkeley Packet Filter [7] commands. The detection engine is programmed using a simple language that describes per packet tests and actions. Its architecture is focused on performance, simplicity, and flexibility. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem. These subsystems ride on top of the

libpcap promiscuous packet sniffing library, which provides a portable packet sniffing and filtering capability. Program configuration, rules parsing, and data structure generation takes place before the sniffer section is initialized, keeping the amount of per packet processing to the minimum required to achieve the base program functionality [8].

1.7 Thesis Outline

In this chapter, some background information about the thesis concept is introduced. In the next chapter, snort which is the case study tool of new multi-threaded design is examined. A flowchart is presented in order to understand algorithmic construction. In the 3^{rd} Chapter, where there is a brainstorming about tuning techniques, implementation alternatives of the new design are discussed. Time profiling results of original snort are obtained and used in order to choose the best alternative. Next Chapter, where a new model is proposed, deals with implementation details of the new design. In the 5^{th} Chapter, test scenarios are devised in order to understand the influence of some parameters on speedup. Next chapter includes the conclusion and tells about future work. And the remainings are the appendix and references.

CHAPTER 2

INTERNAL OF SNORT (Version 1.9.0)

2.1 The Reason for Studying Snort

Snort was designed to fulfill the requirements of a prototypical lightweight network intrusion detection system. It has become a small, flexible, and highly capable system that is in use around the world on both large and small networks [8]. It has attained its initial design goals and is a fully capable alternative to commercial intrusion detection systems in places where it is cost inefficient to install full featured commercial systems.

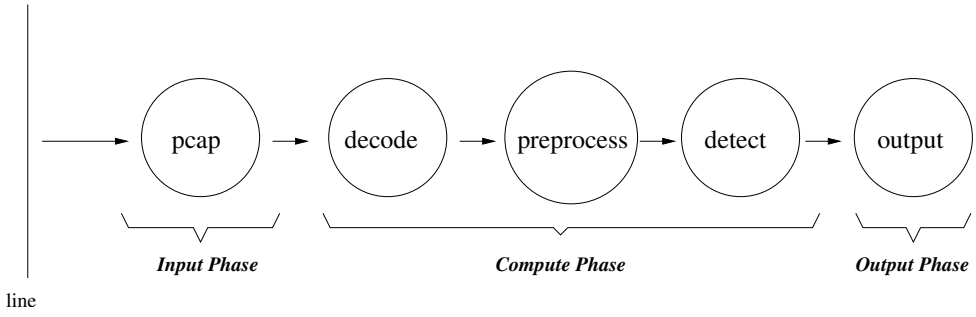

Figure 2.1: Data flow diagram (DFD) of snort

2.2 Data Flow Diagram (DFD) of Snort

The journey of a packet within a snort process is depicted in Fig. 2.1. After a new packet has just arrived, snort fetches it up to the application level by using *libpcap* library. This asynchronous phase corresponds to *input phase* of a typical RBNIDS.

The content of the packet is arranged to be used in only network communication and not readable and processable by snort. *Decode* phase converts the byte sequence of the packet to an understandable form. However, decoding itself is not enough to determine the packet nature. Some packets need further processing in order to be correctly recognized by the rules. This task is performed by *preprocess* phase. After the packet is ready for analyzing, it is compared against the rules for a match, which is the *detect* phase. These three phases compose *compute phase* of a RBNIDS.

When a rule is matched for the packet, the corresponding action in the rule specification is taken. This triggers an outputting process such as logging or alerting, which is performed by *output* phase.

2.3 Main Data Structure on which Snort Algorithm Runs on

As seen from Fig. 2.2, a three dimensional data structure is constructed after the rule files are parsed. 1^{st} dimension is the main rules and it has exactly 5 nodes. The order of the nodes is important since when a new packet is compared against these nodes and their associated rule headers and options, the first node (namely, "pass") is checked first. In the flowchart of the detection algorithm depicted in Fig. 2.3, it is seen that if any rule in this node is matched with the packet, the function, which is responsible for detecting the action with the packet, is returned and the action specified in the node is taken. No remaining nodes in the chain are checked. Therefore, the order of

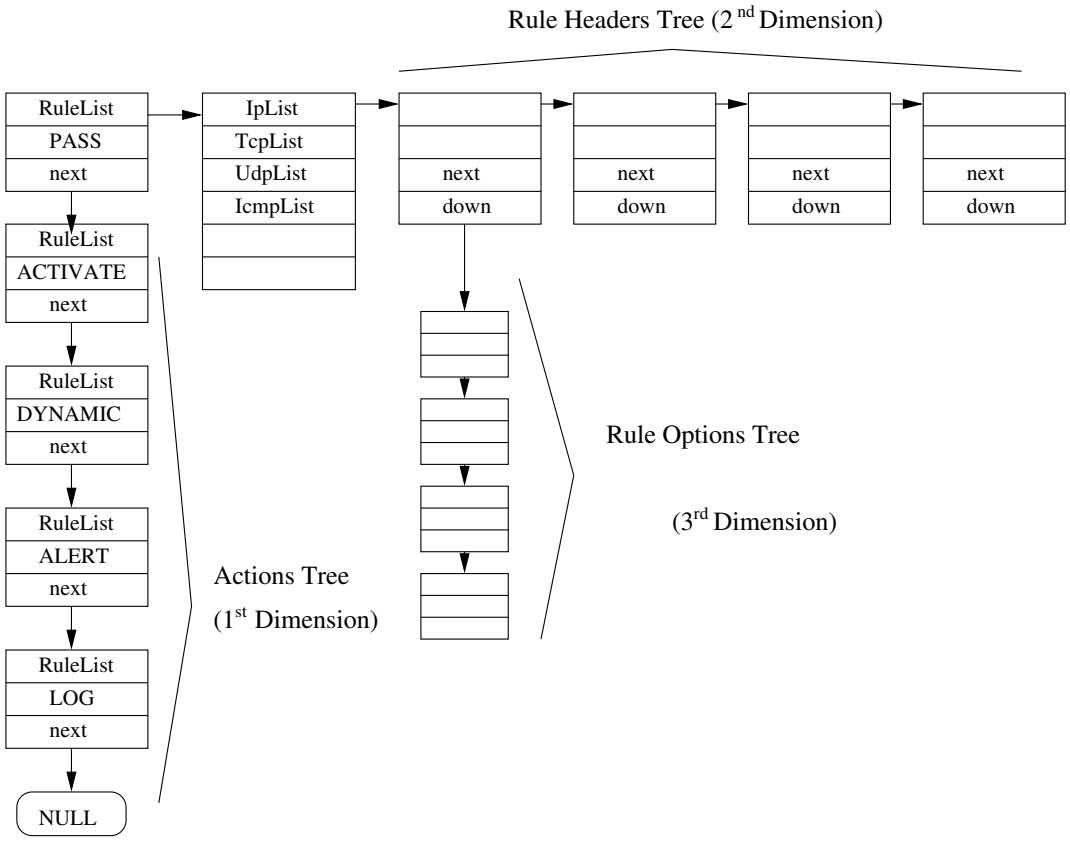

Figure 2.2: Rule data structure of original snort

these nodes is important and changeable by a switch (namely, -o switch, it has two alternatives).

Rule header structures are sorted in ascending order with respect to "destination port". When a packet is examined through the header chains (2^{nd} dimension), the destination ports are compared first. If the packet destination port is smaller than the header structure's destination port, the search in this node is abandoned and other node in the 1^{st} dimension is considered for a possible match, if any.

After a match is found in the 2^{nd}dimension for any node, it is needed that at least one of the option structures of the matched header structure must fit with the packet in order to specify the packet and get the desired action. To find a match in the 3^{rd} dimension, a sequential search is applied.

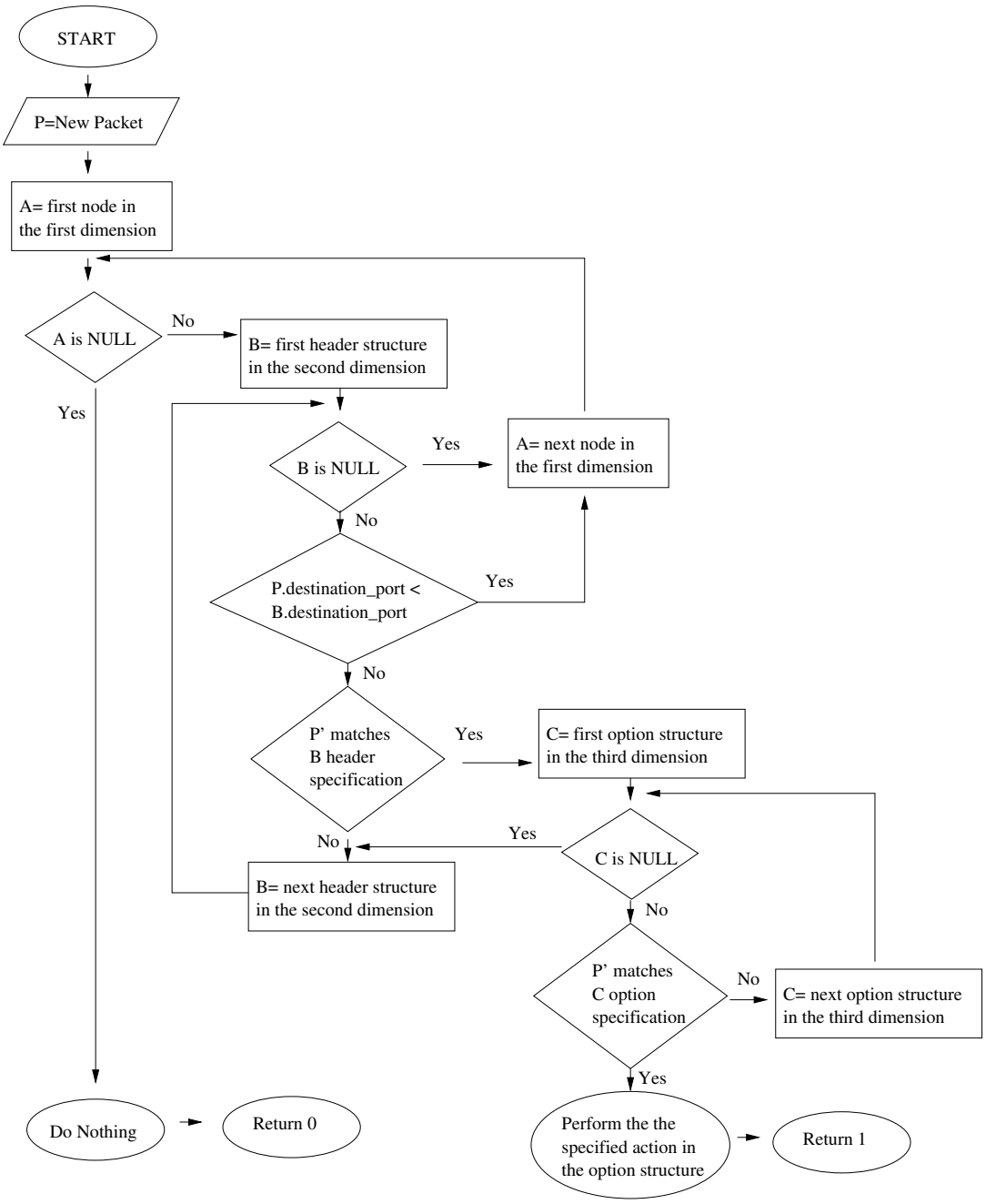

Figure 2.3: Flowchart of the detection algorithm of snort

2.4 Discussion

In section 2.2 on page 11, flow diagram of a packet in snort process is depicted. It is seen that there is no overlapped (pipelined) phases. When one of the phases is executing in one CPU, no other phases execute even if there are more than one CPU. What if there are many rules and some of them are content related? In this situation, detection phase takes too much time to find a match. Although the server has enough processing power (CPUs) to handle all the network traffic, snort may not be schedulable on this server because of its poorly usage of resources.

CHAPTER 3

PERFORMANCE IMPROVEMENT METHODS

In order to discuss improvement methods, it's highly important to measure the time spent in each phase of the snort. The result of the measurement is used to design the optimum concurrency mechanism for the algorithm.

3.1 Time Profiling Snort Phases

In this time profiling process, a sample packet file is created with *tcpdump* program on a highly used server[1]. When tcpdump is collecting packets in a file[2], fragmented packets are sent to the server by using *nmap* scanner tool[3]. The aim is to split up the TCP header over several packets to make the snort busier while preprocessing them. At the same time, *ping flood* attack[4] is performed against the server from another IP[5]. The resultant file contains 216651 packets and 28152 of them (14.935%) are fragmented.

---

[1] IP is 144.122.199.111
[2] tcpdump -w filename
[3] nmap -f -sS -O server-IP
[4] ping -f server-IP
[5] 144.122.3.92

In each sample run sets, there are configuration files. By each configuration, snort is executed several times[6] while replaying the traffic with *tcpreplay* tool and the average value of each set is put into the result. In each run, the environmental settings[7] are kept equal.

Time profiling process consists of three main run sets. Conf.1 through conf.5 (Fig. 3.1)are the sample configurations of run set 1. In this run set, an additional preprocess plugin is added to configuration file[8] in order to understand the preprocessing cost in each configuration. Below, there are five configurations of first run set:

1^{st} **Conf:** content of configuration file is

alert tcp any any -> 144.122.3.92 any (msg: "try1";)

alert icmp any any -> 144.122.3.92 any (msg: "try2";)

alert udp any any -> 144.122.3.92 any (msg: "try3";)

alert tcp any any -> 144.122.199.111 any (msg: "try4";)

alert icmp any any -> 144.122.199.111 any (msg: "try5";)

alert udp any any -> 144.122.199.111 any (msg: "try6";)

2^{nd} **Conf:** 1^{st} configuration plus

preprocessor http_decode: 80 8080 unicode iis_flip_slash iis_alt_unicode full_whitespace

3^{rd} **Conf:** 2^{nd} configuration plus

preprocessor portscan: 144.122.0.0/16 1 1 /var/log/portscan.log

4^{th} **Conf:** 3^{rd} configuration plus

preprocessor stream4: detect_scans, disable_evasion_alerts

5^{th} **Conf:** 4^{th} configuration plus

preprocessor frag2

---

[6] snort -c /etc/snort.conf -n 216651

[7] no other processes are run, the output of each run is deleted, etc...

[8] /etc/snort.conf

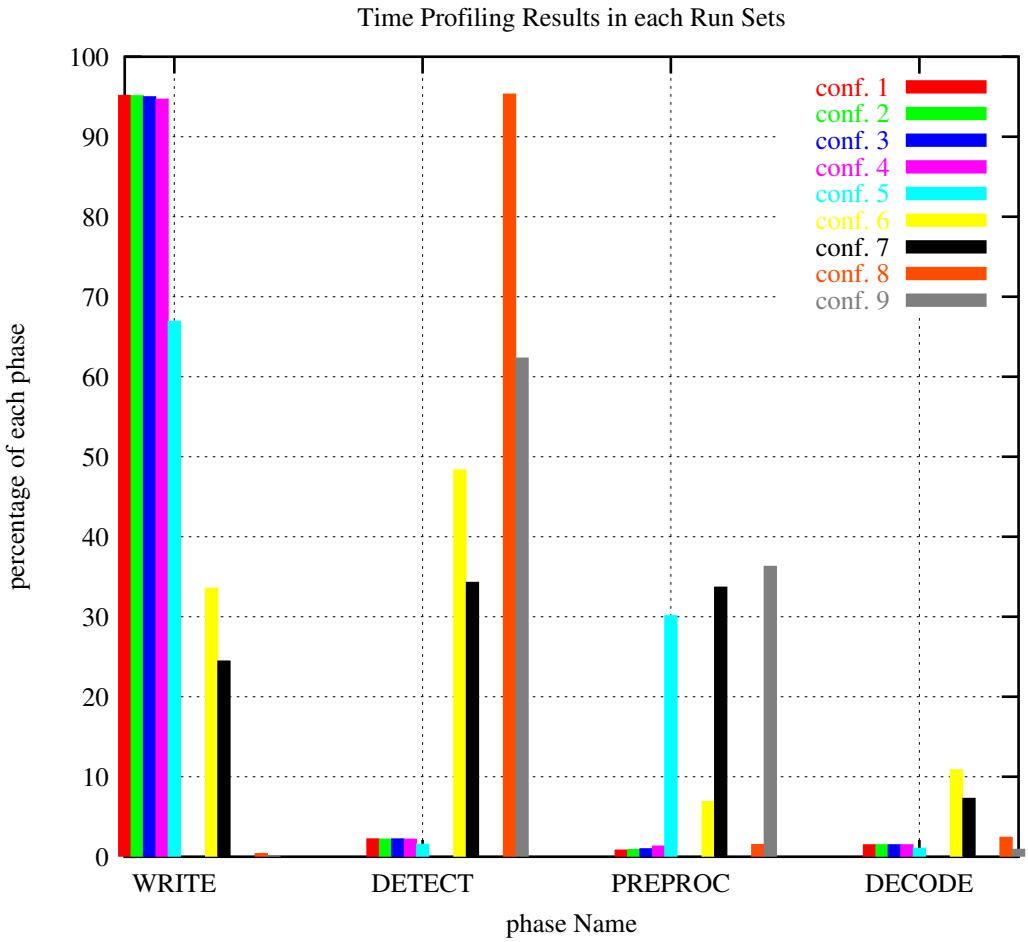

Figure 3.1: Time profiling

The results of sample runs are given in Fig. 3.1. It's easily seen that most time-consuming phase is WRITE with a very big percentage comparing to others, since there aren't many rules. The relative percentage of it decreases with additional preprocessor plugins because of the percentage increase of PREPROCESS phase. It is very noticeable that, at the 5^{th} conf, there is a sharp increase in PREPROCESS phase. The reason is the nature of the packet being replayed. The fifth preprocess keyword added is *frag2* and the replayed file contains many *nmap* and *ping* fragmented packets.

Second run set contains conf.6 and conf7. Conf.7 is a common configuration file[9] that snort uses while protecting a B class[10] network. Conf.6 is just preprocessors extracted. In this run set, it is easily seen that the most time consuming phase of snort is DETECT in an ordinary configuration file.

Third run set contains two hypothetical configurations. Conf.9 contains 4000 rules which are generated by the shell script in Appendix 1. plus 4 preprocessors[11]. Conf.8 is just preprocessors extracted. As seen from the figure, an excessive percentage of DETECT phase is obtained by adding more rules to the configuration file.

When snort is used in front of networks, every addition of a rule introduces some overhead to the DETECT phase. Second run set shows that even a typical configuration file make the DETECT phase most time consuming module. Hence, it is worth hacking through concurrency mechanisms.

3.2 Tuning Alternatives

The program in which the snort algorithm resides is a single thread. Actually, the whole snort software is not single threaded. It has as many threads as the interfaces

[9] actually it's currently used in METU bridge, the details of it are given in Appendix 4
[10] 144.122.0.0/16
[11] just like in conf.5

it actually runs on. For example, if the server it executes on has three interfaces, the running snort has three threads each of which is responsible for each interface and executing the above algorithm. However, this threaded schema isn't for speeding up the algorithm. In our case, it will run on a single interface computer.

In order to obtain maximum speedup, various techniques, some of which are multi-threaded, can be proposed:

3.2.1 Changing the Way of Snort Detection

By changing the detection algorithm, the running time of the detection module can be reduced and throughput of the snort can be increased. For example, after the rule file is parsed, a finite state automata can be generated by grouping similar rules in a single state and detection process is performed between the states of the automata by executing less statements. However, in this technique, the independent parts of the application aren't categorized and resources of the platform aren't fully utilized by these parts. Therefore, this option is not our concern.

3.2.2 Mapping a Group of Rules to a Thread

Each node in the 1^{st} dimension of Fig. 2.2 can be modeled as a single thread. When a new packet is fetched by libpcap, each five threads begin to process it according the their own rule headers and options. In order to decide for a match, the answers form the threads must be fetched. Because the node which is more initial in the 1^{st} dimension is higher priority, the answer of the thread responsible for this node is waited for a decision. When the answers are ready, the action of first successful match (if there are more than one) in the 1^{st} dimension is executed.

In this model (seen in Fig. 3.2), all threads except the successful one steal CPU

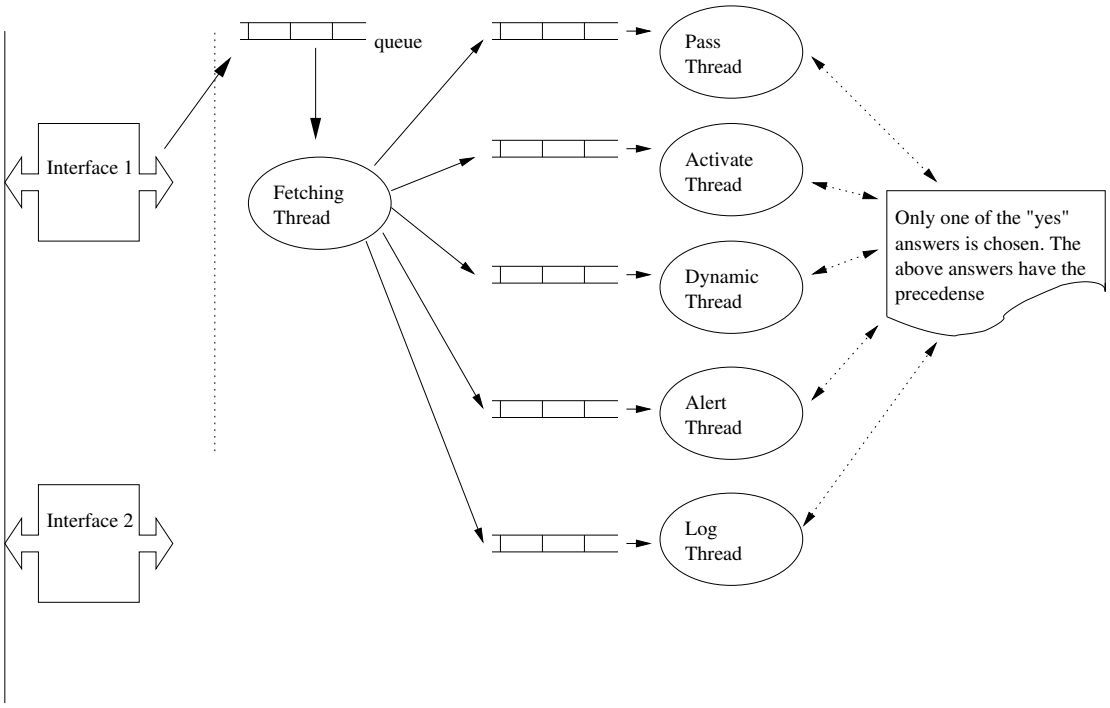

Figure 3.2: Mapping a group of rules to a thread

cycles, since their answers aren't evaluated for the result.

3.2.3 Mapping a Group of Packets to a Thread

As seen from Fig. 3.3, There will be several threads (say them "slave threads", the number of them can be set in the configuration) executing the above detection process. A special thread (say it "fetcher thread" or "dispatcher thread") is responsible for getting packets from the interface and distributing them to the slave threads in a fair manner. The thread with a minimum number of packets in its queue gets the packet.

One disadvantage of this structure is that, each slave thread is unconscious about the nature of the packets it processes. For instance, if an attack is performed with fragmented packets and the packets of this attack are distributed among the slave threads, none of them is capable for a match even if the appropriate rule definition of the attack is specified in the configuration file. In order to recognize such attacks,

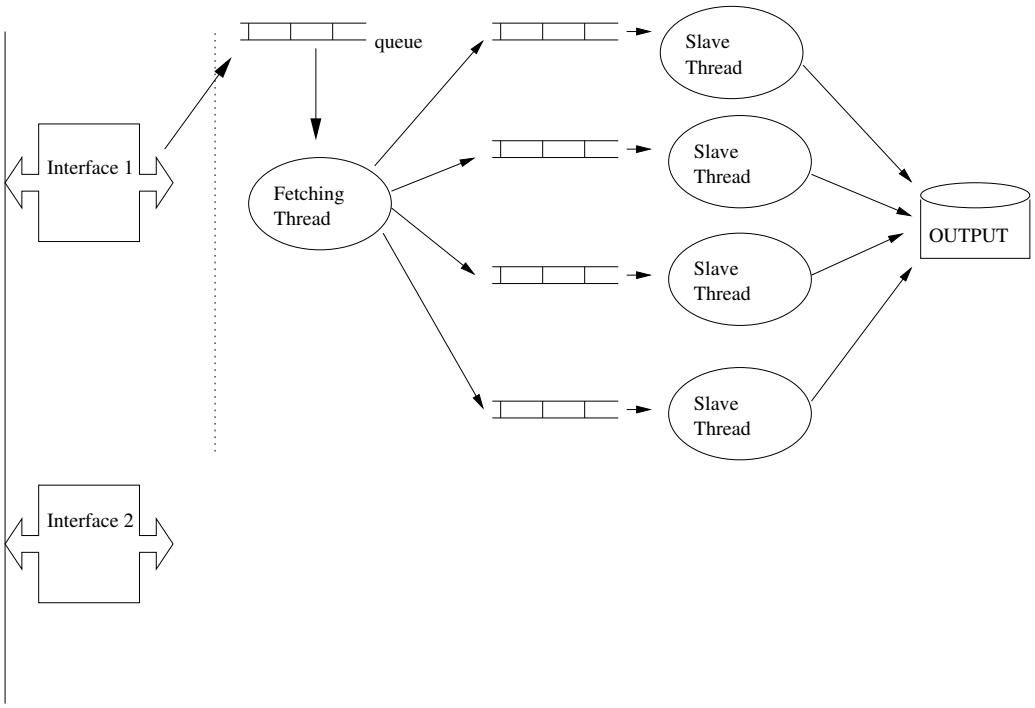

Figure 3.3: Mapping a group of packets to a thread

the related packets must be preprocessed into a single packet prior to delivering the slaves. Hence, the decoding and preprocessing phase of the snort must be in dispatcher thread. This structure is examined in next section.

The other disadvantage is that all the threads are doing the same job. This may lead a race condition when each slave thread attempts to perform an operation on the same single resource at the same time. This condition is highly probable to occur when the tasks of the threads are similar as the one in this structure. For example, at any execution time, most of threads may be making I/O requests. While one of the threads gets the ticket for I/O, the others are blocked. So their executions aren't completely overlapped.

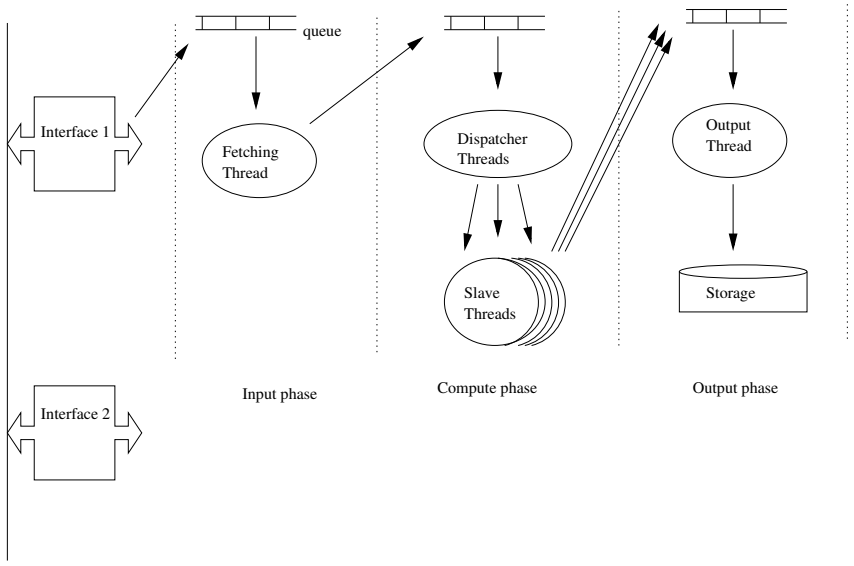

Figure 3.4: Mapping a group of subtasks to a thread

3.2.4 Mapping a Group of Subtasks to a Thread

The computation phase of the snort can be modeled by many threads. As depicted in Fig. 3.4, these threads are fed by an input thread, and the results of detection is processed by an output thread. There are more threads than the CPU numbers in order to keep them all not idle. In the case study, this approach is implemented, tested and compared with original snort.

As explained in the previous section, if different subtasks of snort is fairly deployed between threads, it is more probable that their execution time can be overlapped than dividing similar subtask into threads. While the thread group of one subtask is performing its job using some resources, the thread groups of other subtasks can also perform their jobs using different resources at the same time, because of distinct subtasks they execute. However, it's very crucial to adjust the fairness of task distribution. The profiling results in section 3.1 on page 15 can give us some clues about it. Since the execution time of PCAP and DECODE is very small, their tasks can be merged into one subtask. In section 3.2.3, the obligation of putting PREPROCESS

task in different thread before DETECTION thread is mentioned. Hence, the task of PCAP, DECODE and PREPROCESS can be considered as one subtask.

CHAPTER 4

MULTI-THREADED SNORT

In this chapter, the evolutionary path from the original snort to the multi-threaded architecture is explained. The inherent obstacles originated from operating system limitations such as heap implementation and the nature of modules of multi-thread design are presented. At the end, a feasible infrastructure is proposed and implemented.

4.1 Producer-Consumer Model for Concurrency in Snort

The building blocks of Snort, like other RBNIDS, are input, compute, and output. PCAP library corresponds the input phase. DECODE, PREPROCESS and DETECT simulates the compute phase. OUTPUT describes itself.

The new multi-threaded model works in an assembly line. The output of the first phase (PCAP+DECODE+PREPROCESS) is actually the input of the next phase (DETECT) in the pipeline. As seen from Fig. 4.1, these phases are deployed to three different thread groups. Since the nature of expertise of each thread group is different and they are using different system resources, the chance of occurrence of a bottleneck

for all of them at the same time is very rare. While one group of threads in the pipeline is blocked (doing I/O for example), the other groups of threads are possibly doing their tasks.

4.2 Transforming into Threading

When identifiying and grouping the independent paths of snort algorithm, three main threading groups occupy the main subtasks. The interaction between these thread groups is accomplished with FIFO queues. In order to maintain integrity in these queue structures, POSIX mutexes are embedded in the code although the usage of them introduces some drawback in the efficiency. Mutexes are also needed when there are shared memory blocks entered by different threads.

If the shared memory areas are possible to be owned by threads, they are replicated for each thread by using the utilities of *thread specific data* concept. By this mean, there is a gain in efficiency, instead of creating bottlenecks (with mutexes) when entering these locations.

4.3 Implementation

The infrastructure depicted in Fig. 4.1 is implemented with POSIX thread library. PCAP+DECODE+PREPROCESS, DETECT and OUTPUT modules are three different thread groups and the interaction between these groups is conveyed by two FIFO queues.

4.3.1 PCAP+DECODE+PREPROCESS Thread

In order to understand the evolutionary path into multi-threaded snort, it's necessary to identify the transitional functions between these modules in original snort code.

When a packet is fetched by pcap library *pcap_loop*, *ProcessPacket* function is called for further processing. It, then, calls a function (*grinder*) to decode the packet. The decoded packet is given *Preprocess* function to make it more recognizable by the detection phase. In multi-threaded snort, all the functions up the here is put in the first thread group. The *Preprocess* function is replaced by *Preprocess2* function[1]. The new function calls the preprocess functions as the original one, however it doesn't call *Detect* function. Instead, it puts the packets in a FIFO queue (*requests, a linked list*) by calling *AddItem* function[2]. The first thread group (PCAP+DECODE+PREPROCESS) does this task repeatedly for each packet as a single thread. At this point, it is important to see that the time elapsed to process and identify a single packet in original snort is shifted into this thread. This time has to elapse in order to get next packet. The bigger the time is, the worse the snort efficiency. Since this single thread is ready to get the next packet after adding the previous packet into queue (since, it doesn't have to deal with detection and outputting), an efficiency increase is theoretically expected in multi-threaded snort.

4.3.2 DETECTION Thread Groups

In original code, *Detect* function is responsible for the detection phase. After it traverses the three-dimensional data structure for a possible match, it calls the appropriate output functions. In multi-threaded design, this duty is distributed among a number of (user defined)[3] threads. This threads are implemented by *detect_loop* function[4] which is buried into threads by *pthread_create* function[5]. These threads dequeue

[1] in source file detect.c
[2] in source file detect.c
[3] in mythread.h file, NUM_HANDLER_THREADS
[4] in source file detect.c
[5] in source file snort.c

the packet records from the queue by *get_request*[6] and do its task by *handle_request*[7] functions. When there is no packet in the queue, the detection threads wait on a condition variable[8] to be signaled, hence they steal no CPU cycles. As soon as a new packet is queued, the first thread signals the condition variable and one of the detection threads wakes up and performs its task.

4.3.3 OUTPUT Thread

Instead of calling output functions, the *detect_loop* adds the output records into another FIFO queue[9] by calling *AddItem2* function[10]. The output module, which is created as a thread[11] and realized by the function *output_loop*[12], is responsible to take the appropriate actions. This thread dequeues the output records from the queue by *get_request2*[13] and does its task by *handle_request2*[14] functions. When there are no output records in the queue, the behavior of output thread is the same as detection threads.

4.4 Comparing the Outputs

Since the packet process order of the new design is non-deterministic, the outputs of it might be seen different from the original one. Therefore, in order to compare the both results for exact match, a mechanism is to be found. The program in Appendix 2 is written to make the comparison. It first sorts[15] each output file of both version

---

[6] in source file detect.c
[7] in source file detect.c
[8] got_request
[9] requests2
[10] in source file detect.c
[11] in source file snort.c
[12] in source file detect.c
[13] in source file detect.c
[14] in source file detect.c
[15] linux command tool, sort

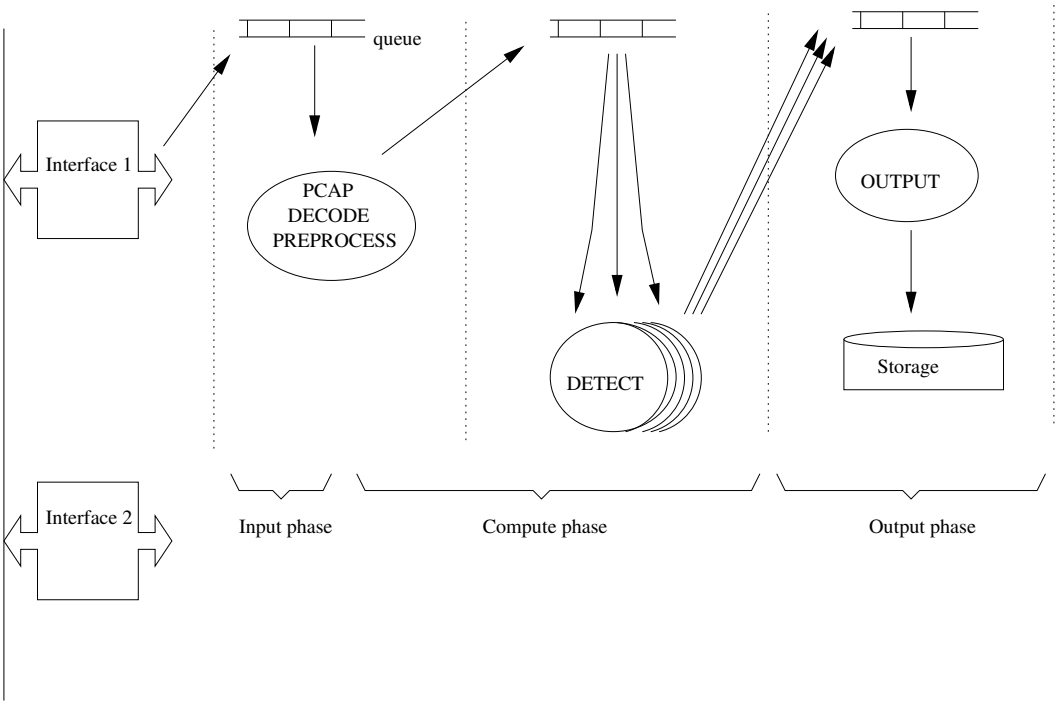

Figure 4.1: Multi-threaded overview of producer-consumer model

and then compares them. It is seen that there is no differences in the output.

4.5 Implementation Environment

4.5.1 "Kernel Space Threads" vs. "User Space Threads"

The multi-threaded snort is developed in Linux[16] operating system. The running kernel is 2.4.23[17] and the thread library is LinuxThreads (version 0.9). In Linux Threads Implementation, each thread is a separate process. The kernel scheduler itself schedules the threads just like Unix processes [9], which supplies the main strength that it fully utilizes any multiprocessors architecture. Additionally, by shifting the scheduling job from thread library to kernel, it also results in a simpler, more robust thread library, especially with respect to blocking system calls [10]. But this also introduces the disadvantage that it produces more overhead to make context swithces

[16] debian 3.0, debian.org
[17] from kernel.org

on mutex and condition operations since these must go through kernel functions. This disadvantage is overcome by the fact that, in Linux, context switching algorithm runs very efficiently [11]. The difference with respect to the processes is, however, they share the same address space with each other through *clone()* system call.

There are basically two other models. The "many-to-one" model, in which context switching is handled by thread library (using a user-space scheduler), kernel sees one process running that contains all the threads. This model can not be an alternative to discuss since it does not take advantage of multiprocessors, and it is almost impossible to handle blocking I/O operations properly. Although there are several user-level thread libraries available for Linux, they are deficient in functionality, performance, and/or robustness [11].

The advantages of both kernel-level and user-level scheduling are merged into the "many-to-many model": while there are several kernel-level threads running concurrently and fully taking advantage of SMP architecture, each of them executes a user-level scheduler that selects between user threads in it. Most commercial Unix systems (Solaris, Digital Unix, IRIX) implement POSIX threads this way. This model is designed by being inspired of both infrastructures of the "many-to-one" and the "one-to-one" model, and is attractive because it avoids the worst-case behaviors of both models – especially on kernels where context switches are expensive, such as Digital Unix. Unfortunately, it is pretty complex to implement, and requires kernel support which Linux does not provide. Linus Torvalds and other Linux kernel developers have always been pushing the "one-to-one" model in the name of overall simplicity, and are doing a pretty good job of making kernel-level context switches between threads efficient. LinuxThreads is just following the general direction they set [11].

4.5.2 Linux Kernel - 2.4 vs 2.6

With respect to threads and scheduling view, the major improvement of new kernel is its scheduling algorithm. Unlike 2.4 having the schedular of O(n) complexity, new kernel has a schedular of **O(1)** complexity that uses two sets of priority arrays to achieve fairness. That results a good schedular for wakeup, context-switching and timer interrupt overhead [12].

The consequences of O(1) schedular, as its name tells, it scales well regardless of how many processes is waiting in the run queue. Secondly, since the core design of the algorithm is SMP-aware, scheduling on that architecture got improved significantly. That brings better scalability and performance results [12]. These enhancements are enough to understand the advantages of the new scheduler. A typical schedular usually has the most work to do when there are more tasks in the systems. Since O(1) decides the task to run in a constant time period, these kinds of heavy-loaded situations do not bother the efficiency of it. Contrarily, these are the best conditions that kernel 2.6 has better throughputs.

4.6 Some Drawbacks of the Implementation

The methodologies used in converting original snort to multi-threaded one are described in section 4.2. When applying these methodologies, not all the features of original snort are exported to the new one. Hence, multi-threaded snort has less features than the original one. For instance, *stream4* preprocessor can not be used in the configuration file in the new system. However, for the sake of good comparison, many of the preprocessors like *http_decode*, *portscan*, etc. are exported. Since both version run with the same configuration file (of course with exported preprocessors),

the results give us a clue about how the new design speedups the system.

Since there is potential of contention inside the *free()* function in Linux Heap Implementation, there were some **mandatory** memory leaks in the old design of multi-threaded snort. The current heap implementation uses multiple independent sub-heaps called arenas. For the sake of concurrency protection, each arena contains its own mutex. If there are enough arenas in a process heap structure and there exists a schema to distribute these structures evenly between the threads of the process, then there is less risk for the contention for the arenas. The working mechanism of this fair schema is: in malloc(), a test is made in order to acquire the mutex of an arena. Since *trylock()* function is used, the calling process does not hang and continues until an arena having an unlocked mutex is found. If there aren't such arenas, a new arena is created with a locked mutex and given to the calling process [13].

On the contrary, when a process tries to free a block of memory, the heap implementation puts a performance barrier into the scene. As a design requirement, a block of memory that is freed must be turned back to the arena where it was originally allocated. When a process wants to free a block, it firstly desires to lock the mutex of the owning arena. If it is locked by another process, the calling process has to wait until the lock is over on the mutex since it can not return the memory block back to another arena. This scenario is usually obtained in multi-threaded applications where one group of threads is busy with malloc() while other threads are doing free() [13]. Since this feature is in our multi-threaded design, it is obvious to be introduced with performance decrease while calling free(). In order to overcome this condition, free() statements wouldn't be used since this would cause memory leaks. Hence, a different solution is to be found.

Consequently, it is decided that the queues between the thread groups aren't dynamically re-sizable. They have static length and the records going in and out are inserted into the first slot that isn't used. However, since the output thread is much slower than the input thread, the queue structures are easily filled up in initial seconds of the test runs. Because input thread has to wait for the output thread activities to empty a slot in order to get the next packet. This dependency means worse performance even than original snort. Because of the limitation of linux heap implementation and inherent slowness of output thread, a limited version of the static-size queue structure is devised. The queues are allocated as long as possible and not much packets exceeding the queues length are used.

4.7 Portability of Multi-threaded Design

Currently, the latest stable version of snort is 2.0.1. When the structure of this version is examined, it is seen that it also has the same modules as the version 1.9.0. However, there are some major code changes in some modules[18], and this requires much effort to apply the design to the new version.

[18] they can be traced from snort.org

CHAPTER 5

TEST RESULTS and COMPARISON

5.1 Test Media

The above design is implemented using snort (version 1.9.0) source tree. As shown in Fig. 5.1, the new application has four threads, one for input, two of them for detect and last one is for output. Original snort and the new multi-threaded snort is executed on a two-processors PC[1]. The operating system is Debian(3.0) and it contains a linux kernel version of 2.4.23[2].

As in the Fig. 5.2, the test computer is connected to another computer (console) with a cross cable. The packet file, which is obtained from METU bridge using tcpdump program[3] and contains 500000 packets, is replayed from the console PC with *tcpreplay*[4] tool. Without no interruption, maximum speed that this tool can reach over the cross cable is 45 Mbps. This speed is experimentally obtained after a few tries and in order to reach it, some parameters on /proc file system have to be

---

[1] Pentium-III/1133 Mhz, 926MB memory

[2] from kernel.org

[3] tcpdump -i <interface> -w <output file> -c <packet count>

[4] version 1.4.6 that is compiled against libnet (version 1.1.1) [14] and libpcap (version 2003.12.22) [15]

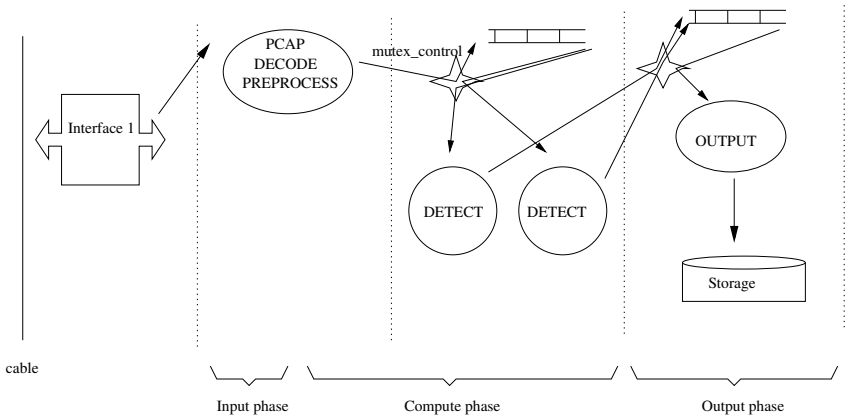

Figure 5.1: The layout of threads and their connectivity

tuned. The script that is responsible for tuning is Appendix 5.

5.2 Testing with Seven Different Configuration Files

In order to understand to effect of configuration file, seven different contexts are composed for the test. The contents of these files:

1^{st} **Conf:** content of configuration file is

pass tcp any any -> 144.122.3.92 any (msg: "try1";)

pass icmp any any -> 144.122.3.92 any (msg: "try2";)

pass udp any any -> 144.122.3.92 any (msg: "try3";)

pass tcp any any -> 144.122.199.111 any (msg: "try4";)

pass icmp any any -> 144.122.199.111 any (msg: "try5";)

pass udp any any -> 144.122.199.111 any (msg: "try6";)

2^{nd} **Conf:** content of configuration file is

alert tcp any any -> 144.122.199.12 80 (msg: "try1";)

alert icmp any any -> 144.122.3.92 any (msg: "try2";)

alert udp any any -> 144.122.3.92 any (msg: "try3";)

alert tcp 144.122.0.0/16 any -> any 80 (msg: "try4";)

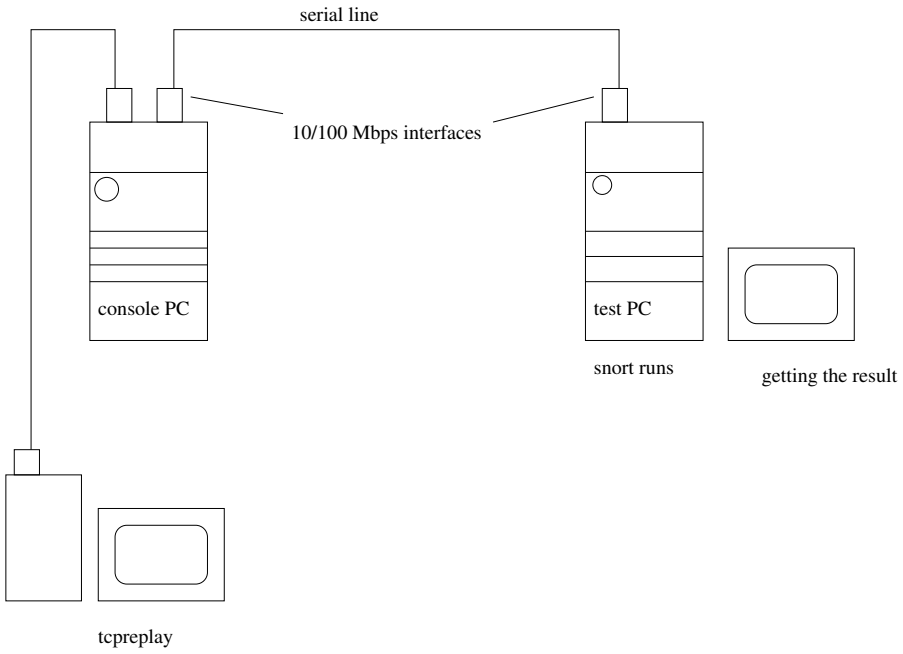

Figure 5.2: An overview of test computers and their connections

alert icmp any any -> 144.122.199.111 any (msg: "try5";)

alert udp any any -> 144.122.199.111 any (msg: "try6";)

preprocessor http_decode: 80 8080 unicode iis_flip_slash iis_alt_unicode full_whitespace

preprocessor portscan: 144.122.0.0/16 1 1 /var/log/snort/portscan.log

3^{rd} **Conf:** METU configuration file [5]

4^{th} **Conf:** 3^{rd} Conf + 50 content matching rules[6]

5^{th} **Conf:** 3^{rd} Conf + 100 content matching rules

6^{th} **Conf:** 3^{rd} Conf + 500 content matching rules

7^{th} **Conf:** 3^{rd} Conf + 1000 content matching rules

In the first test set, the packet file is replayed[7] from the console while both snort versions run on the test PC exclusively. The aim of this test is to determine maximum

[5] Details of this file are given in Appendix 4
[6] these rules are randomly generated using the C program in Appendix 3
[7] tcpreplay -i <interface> -r <speed> <packet file name>

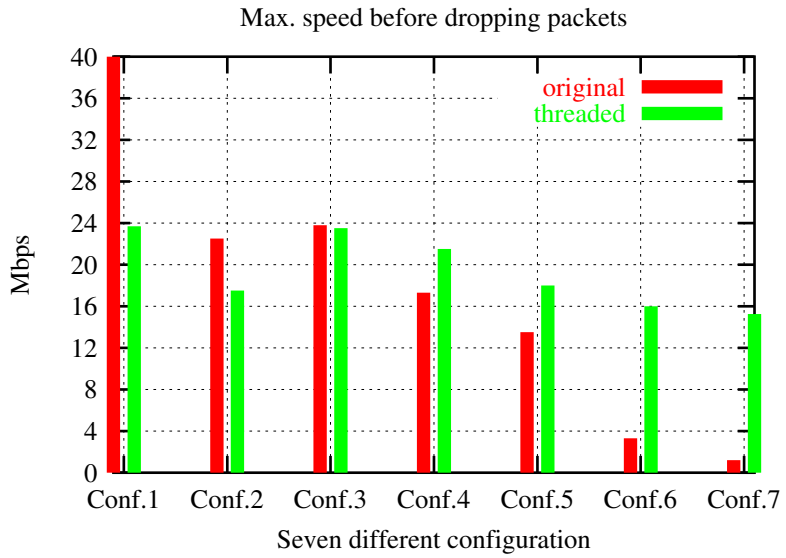

Figure 5.3: Maximum speed of packet sent before dropping begins

packet speed that both versions of snort can handle.

As seen from the figure 5.3, performance of the multi-threaded snort is worse than the original one in first two configurations. In other words, the original snort is capable of handling more packets than the multi-threaded one before packet dropping begins. The reason is, as seen from the content of configuration files, the duty of detection phase is very small. Since there exists some overhead caused by schemes such as mutexes and context switching, multi-threaded snort is obviously beaten. However, as the content of the configuration file gets larger, the new snort starts to show better performance. Through Conf.3 to Conf.7, the dramatic increase in handling capability of multi-threaded design is easily seen.

Figure 5.4 depicts the comparison between running times of both snort when they are executed using packets that are fed from a local file. The aim of the test is to demonstrate the speedup gain in new design. In the runs of both version, packets are captured from a local file.[8]

[8] with -r parameter

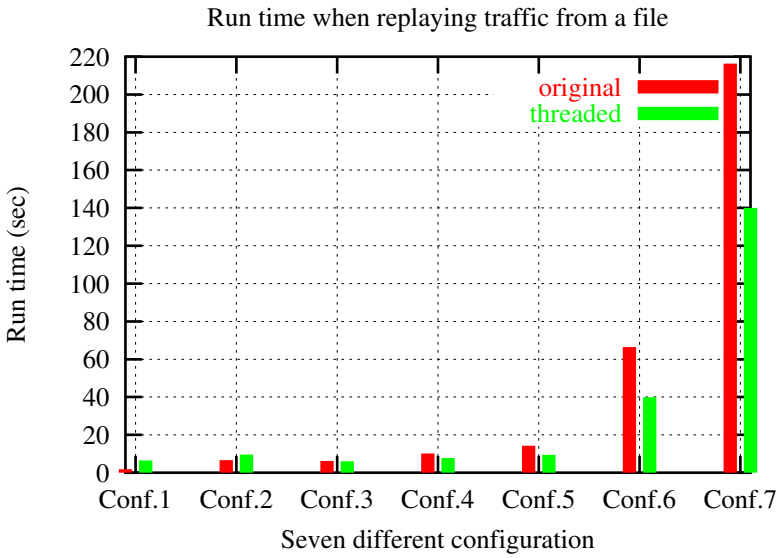

Figure 5.4: Running time of both snort when they capture packets from a local file

As in the figure 5.3, first two configurations draw a profile in behalf of the original snort. However, the multi-threaded snort runs faster than the original one in other configuration files. The more additional rules are added into configuration, the more speedup is gained. The performance difference between each version gets larger when duty on the detection phase increases. From the figure, when going left to right on the *x axis*, a slight convergence to *perfect speedup*[9] is easily realizable.

5.3 Throughput Difference between Kernel 2.4 and Kernel 2.6

In Fig. 5.5, multi-threaded snort is run on each of kernel 2.4 and 2.6. The aim is to compare the new kernel version with respect to thread scheduling algorithm. Through configuration 3 to configuration 7, additional content matching rules cause the processing task of detection thread to be very loaded. Depicted in the figure that the more duty these threads have, the more efficient the kernel 2.6 runs the multi-threaded snort. Since kernel 2.6 handles extreme loads more smoothly without

[9] that is two, since the PC has two processors

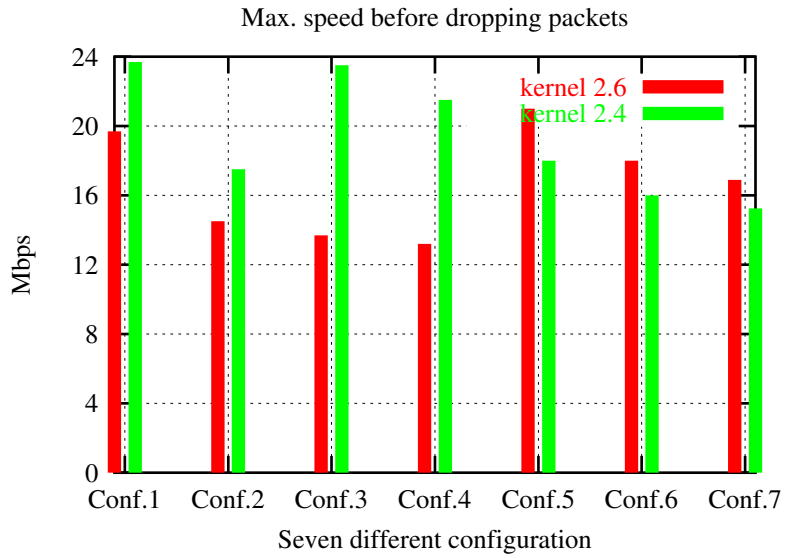

Figure 5.5: Performance comparison of kernel version 2.4. and 2.6 while running multi-threaded snort

breakdown and scheduling storms [16], it shows its power, which is mainly caused by O(1) scheduler, when there are high loads. As also seen from Fig. 5.6, the multi-threaded snort runs faster in kernel 2.6 than in kernel 2.4 when load is increased. In the chart, load values are multiplied by 10 in order to make them realizable. It shows that the new kernel scales well especially when there are high loads.

5.4 Influence of Thread Number on Speedup

In Figure 5.7, the maximum speedup is reached when there are more threads than number of CPUs. The reason is, when some of the threads are blocked for making I/O, for example, other threads should keep all CPUs busy in order to get better speedup. On the other hand, when there are much more threads than the CPU number, overhead cost beats the speedup gain. Therefore the number of threads has to be a bit more than the CPU numbers, not much more.

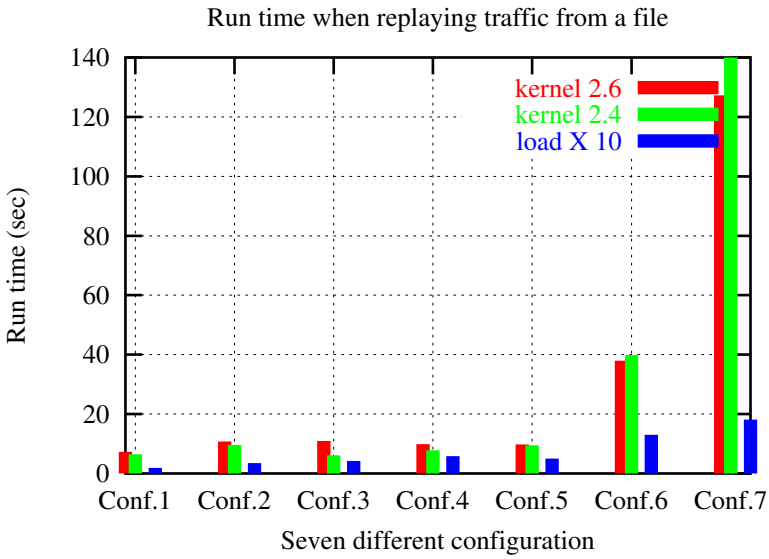

Figure 5.6: Kernel 2.4 vs. 2.6 when packets are captured from a local file

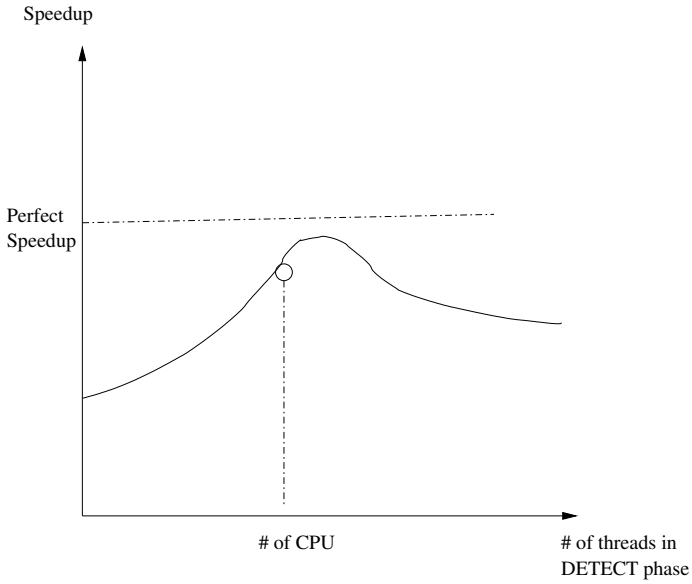

Figure 5.7: Influence of thread number on speedup

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this study, RBNIDS is analyzed with respect to concurrency paradigm. Snort, as a widely used RBNIDS, is selected for proving the proposed statement. It is the premier work that snort execution paths are deeply identified, documented, and a structural multi-threaded schema is designed for its algorithm.

In the implementation, DETECTION phase is recoded with some thread library functions since it is the most time consuming module of the snort algorithm. The results from the tests tell us, by re-implementation of an ordinary sequential RBNIDS, a pretty nice speedup is easily obtained even if the execution platform is limited on resources.

The main goal of the design is multiplexing the detect phase of the snort in order to make it more responsive the pending packets. When adding more duty to the detect phase, it is seen that the performance gain between two models increases and this shows the success of the design.

In the test results, it is seen that kernel 2.6 schedules threads better than kernel 2.4 when there is more load on the system. The reason for this scability is the new schedular algorithm of complexity O(1) instead of old O(n).

The proposal and the results show that there are independent paths in RBNIDSs and these can easily execute concurrently on processors, meaning very important gains in speedup. The consequences of the study are good focal points for such systems.

6.2 Future Work

In the profiling and test sections, figures show that original snort spends more time on output module when there is an extreme demand on disk access. This introduces performance impact on systems where disk speed is too slow. It may be another study to speed up the outputting. Instead of a one thread and local disk access, the output phase can be modeled with parallel threads which are responsible for inserting records into the remote databases.

In order to eliminate memory leaks and reduce the overhead on two queue data structures, a different multi-threaded design is used. In the implementation, the queues are allocated first. It is almost as much as the physical memory lets. However, it has some drawbacks as explained in the implementation chapter. Instead of this, a different mechanism such as PIPE[1] structure may be used.

As a future work, the proposed infrastructure is planned to be designed as "distributed processes" in a cluster systems rather than SMP machines. Since clusters are very cheap to buy and scale up, any performance improvements on these systems are well appreciated.

[1] unix PIPEs

APPENDIX A

1. Shell script to produce configuration file in profiling

```
for j in ‘seq 110 120‘; do
for k in ‘seq 21 100‘; do
    printf ”alert tcp 144.122.3.81 any -> 144.122.199.%d %d (msg:%d %d ;)\n” $j $k
$j $k
    printf ”alert tcp 144.122.3.92 any -> 144.122.199.%d %d (msg:%d %d ;)\n” $j $k
$j $k;
    printf ”alert tcp 144.122.148.0/24 any -> 144.122.199.%d %d (msg:%d %d ;)\n”
$j $k $j $k;
    printf ”alert tcp 144.122.149.0/24 any -> 144.122.199.%d %d (msg:%d %d ;)\n”
$j $k $j $k;
    printf ”alert tcp 144.122.150.0/24 any -> 144.122.199.%d %d (msg:%d %d ;)\n”
$j $k $j $k;
    printf ”alert tcp 144.122.151.0/24 any -> 144.122.199.%d %d (msg:%d %d ;)\n”
$j $k $j $k; done
```

done

2. Shell script to compare outputs of both version

for i in ‘find $1 -type f | grep -v out$‘; do sort $i > $i.out; done

for i in ‘find $2 -type f | grep -v out$‘; do sort $i $i.out; done

for i in ‘find $1 -type f | grep out$ | cut -d’/’ -f2-‘; do echo $i; diff $1/$i $2/$i; done

3. Source code of the C program in order to ramdomly generate context matching rules

```
void main(int argc,char **argv)
{
char line[5];
int i,j;
line[4]='\0';
for(i=0;i<atoi(argv[1]);i++) {
    for(j=0;j<4;j++) line[j]=65+(int) (25.0*rand()/(RAND_MAX+1.0));
    printf("alert tcp any any -> any any (msg: \"alarm%d!\"; content: \"%s\";)\n",i,line);
}
```

4. Categorization of snort configuration file that is used in METU campus

It has 2077 alert rules in 48 different files. These files are downloaded from snort home page [17] and customized for the campus. The rule distribution if these files are:

attack-responses.rules 15

backdoor.rules 57

bad-traffic.rules 14

chat.rules 18

ddos.rules 33

deleted.rules 216

dns.rules 19

dos.rules 18

experimental.rules 0

exploit.rules 36

finger.rules 13

ftp.rules 52

icmp-info.rules 93

icmp.rules 19

imap.rules 16

info.rules 7

local.rules 30

misc.rules 40

multimedia.rules 6

mysql.rules 2

netbios.rules 18

nntp.rules 2

oracle.rules 25

other-ids.rules 3

p2p.rules 15

policy.rules 22

pop2.rules 4

pop3.rules 19

porn.rules 21

rpc.rules 128

rservices.rules 13

scan.rules 24

shellcode.rules 19

smtp.rules 25

snmp.rules 17

sql.rules 43

telnet.rules 14

tftp.rules 9

virus.rules 19

web-attacks.rules 47

web-cgi.rules 344

web-client.rules 6

web-coldfusion.rules 35

web-frontpage.rules 34

web-iis.rules 111

web-misc.rules 292

web-php.rules 62

x11.rules 2

5. The shell script to make /proc file system enhancements

echo "32768 65535" > /proc/sys/net/ipv4/ip_local_port_range

echo 0 > /proc/sys/net/ipv4/tcp_timestamps

```
echo 0 > /proc/sys/net/ipv4/tcp_sack

echo 0 > /proc/sys/net/ipv4/tcp_window_scaling

echo 131056 > /proc/sys/net/ipv4/ip_conntrack_max

echo 524286 > /proc/sys/net/core/rmem_default

echo 524286 > /proc/sys/net/core/rmem_max

echo 524286 > /proc/sys/net/core/wmem_default

echo 524286 > /proc/sys/net/core/wmem_max

echo 32768 > /proc/sys/fs/file-max

echo 67108864 > /proc/sys/kernel/shmmax

echo 256 > /proc/sys/net/ipv4/neigh/default/gc_thresh1

echo 1024 > /proc/sys/net/ipv4/neigh/default/gc_thresh2

echo 2048 > /proc/sys/net/ipv4/neigh/default/gc_thresh3
```

REFERENCES

[1] T. F. Lunt, *A Survey of Intrusion Detection Techniques*, ch. 12, pp. 405–418. Computers and Security, June 1993.

[2] Anonymous, "Maximum linux security,", 1999, `http://mandrake.petra.ac.id:8888/info/max/BkPg155x247.htm` .

[3] G. R. Andrews, *Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[4] K. Ilgun, "Ustat: A real-time intrusion detection system for unix," (Oakland, California, Los Alamitos), pp. 16–28, Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, May 1993.

[5] A. S. Tanenbaum, *Modern Operating Systems.* Prentice Hall, 2 ed., 2001.

[6] V. Jacobson, C. Leres, and S. McCanne, "libpcap," Lawrence Berkeley National Laboratory, 1994. http://www-nrg.ee.lbl.gov.

[7] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," USENIX Technical Conference Proceedings, 1993.

[8] M. Roesch, "Lightweight intrusion detection for networks,", 2003, `http://www.snort.org/docs/lisapaper.txt` .

[9] X. Leroy, "Linuxthreads - posix 1003.1c kernel threads for linux,", 1996, `http://pauillac.inria.fr/~xleroy/linuxthreads/README` .

[10] Xavier.Leroy@inria.fr, "The linuxthreads library,", 2003, `http://pauillac.inria.fr/~xleroy/linuxthreads/` .

[11] Xavier.Leroy@inria.fr, "Internals of linuxthreads,", 2003, `http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html` .

[12] J. Andrews, "Interview: Ingo molnar,", 2002, `http://kerneltrap.org/node/view/517` .

[13] D. Boreham, "Linux heap, contention in free(),", 2003, `http://www.bozemanpass.com/info/linux/malloc/Linux_Heap_Contention.html` .

[14] "Libnet packet assembly,", 2003, `http://www.packetfactory.net/libnet` .

[15] "Tcpdump public repository,", 2003, `http://www.tcpdump.org` .

[16] J. Andrews, "Linux: New scheduler proposal/match,", 2002, http://kerneltrap.org/node/view/341 .

[17] "Snort.org,", 2003, http://www.snort.org .