國 立 交 通 大 學

電機與控制工程學系

碩士論文

利用多核心處理器平台平行處理網路入侵偵測系統

A Load-balanced Parallel Architecture for Network Intrusion Detection System

研 究 生：陳柏廷

Student: Bo-Ting Chen

指導教授：黃育綸　博士

Advisor: Dr. Yu-Lun Huang

中華民國九十八年五月

May, 2009

利用多核心處理器平台平行處理網路入侵偵測系統

A Load-balanced Parallel Architecture for Network Intrusion Detection System

研 究 生：陳柏廷　　Student: Bo-Ting Chen

指導教授：黃育綸　博士　　Advisor: Dr. Yu-Lun Huang

國 立 交 通 大 學

電機與控制工程學系

碩士論文

A Thesis

Submitted to Department of Electrical and Control Engineering

College of Electrical Engineering

National Chiao Tung University

in partial Fulfill of the Requirements

for the Degree of

Master

in

Department of Electrical and Control Engineering

May, 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年五月

利用多核心處理器平台平行處理網路入侵偵測系統

學生：陳柏廷　　指導教授：黃育綸　博士

國 立 交 通 大 學電機與控制工程學系（研究所）碩士班

摘　　要

網路入侵偵測系統(NIDS)常用於監控企業內部之網路與偵測來自外部的攻擊行為。NIDS在執行封包分析時是非常消耗計算資源的，然而面對不斷增加的網路流量，傳統的單執行緒NIDS遭遇到無法完全發揮多核心處理器效能的困境。在這篇論文中，我們提出了一個多執行緒的NIDS架構(以下簡稱為bmtNIDS)，使多核心處理器中所有的計算資源能有效運用在攻擊行為的偵測上，並藉以提高封包的處理量。bmtNIDS允許所有的執行緒同時接收封包，並利用封包過濾器避免兩個執行緒擷取到相同的封包。在此架構中，我們克服了傳統單執行緒NIDS在執行時必須依照接收封包的順序執行分析的規範，並適度的減少了在資料存取時必須被同步機制所保護的資料結構數量來提升系統的整體效能。除此之外，我們也設計了一個被動式的負載平衡機制，根據每個核心的使用量，動態地決定處理新封包的執行緒。如此，可以避免NIDS將封包分析的工作過度集中於某些特定執行緒上，進而導致作業系統丟棄來不及處裡的封包。根據我們在四核心機器上的實驗結果發現：(1)在300Mbps的傳輸速率下，bmtNIDS提高了Snort的效能約1.5倍；(2)相較於他人的多執行緒NIDS，我們也提高了10%的網路封包分析率；(3)bmtNIDS提供一個較好的資源使用方式，使NIDS效能不因其他計算需求量大的應用程式而受到影響。

A Load-balanced Parallel Architecture for Network Intrusion Detection System

Student: Bo-Ting Chen Advisor: Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

National Chiao Tung University

Abstract

In this thesis, we propose a balanced multi-thread NIDS, bmtNIDS, to get a better effeciecncy when running in a multi-core system. bmtNIDS supports multiple threads for simultaneous packet captures, such a design benefits from reducing data migrations between threads. To prevent threads from receiving duplicate packets, bmtNIDS uses a kernel traffic splitter to distribute packets among threads. Since packets are distributed based on flows, bmtNIDS performs access synchronization only on tables recording information between flows, and thereby access synchronizations can be dramatically reduced. In addition, a passive load balancing (PLB) algorithm is proposed to distribute workloads by CPU utilizations, rather than just counting the number of buffered packets. Compared to the conventional load balancing algorithm, bmtNIDS/PLB improves the packet inspection ratio by 10%. In this research, we realize bmtNIDS on Snort and conduct a series of experiments to compare the performance between exsiting multi-thread NIDS systems. From the experiment results, bmtNIDS has an improvement by a factor of 1.5 if the packet transmission rate is higher than 300Mbps. bmtNIDS also has a better resource untilization stratgy, and hence the performance of bmtNIDS is not affected if the system also runs a computing-intensive application.

誌謝

在這裡首先要感謝我的指導老師，黃育綸博士，在這兩年內認真和不怕麻煩的教導我，包容了我的無知與錯誤，一步一步帶我跨過許多研究的關卡。 也很感謝iCAST (The International Collaboration for Advancing Security Technology)計劃提供了一個很棒的機會，讓我能夠赴美國加州柏克萊分校進行半年的合作研究。 在此期間，我要特別感謝Prof. Vern Paxson，Prof. Doug Tyger和研究員Robin Sommer的指導，以及邱建樺學長和蕭宇凱學長的題點，沒有你們，這篇論文無法順利完成。 也很開心能和SWOON平台的新竹交大研發團隊成為好朋友，經過100b次測試的平台一定是沒問題的。 除此之外，我要感謝黃詠文學長和蔡欣宜學姐在系統實作以及論文寫作上的大力協助，讓我能夠把一個月當成一年來用。 最後感謝RTES Lab的全體，實驗室的牌咖是研究苦悶時最好的消遣，我愛你們。

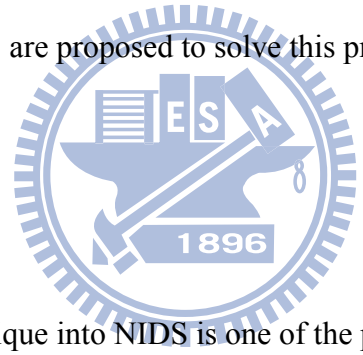
Contents

List of Figures

Chapter 1

Introduction

Resource consumption is one of the major concerns when designing a network intrusion detection system (NIDS), since a NIDS normally performs some computing-intensive operations to exhaust the computing resources in a system. However, the performance advancement on a single core processor can not keep up with the demand of NIDS, even the evolution of a processor reaches the ultimate limits of Moore's Law. To meet the increasing resource requirements and network traffic, parallel NIDS are proposed to solve this problem.



1.1 Background

Introducing cluster technique into NIDS is one of the parallelisms of NIDS designs. NIDS cluster uses a powerful front-end device as network tap and distributes packets to multiple individual hosts, each runs a NIDS process on it [1, 2]. Building up a NIDS cluster brings the possibility to handle high speed network traffic by simply extending its scale, but splitting traffic among hosts could cause a correlation information lost (e.g. scan attack detection).

Another solution to parallelizing NIDS designs is to leverage multi-core processors of a modern computing system. With multi-cores, a NIDS system may perform analysis on multiple packets simultaneously. Since a multi-core processor can be simply treated as an aggregation of multiple hosts, so it can ideally provide a N times computing ability than a single host, where N is the number of logical computing units in a multi-core processor. Since traditional NIDS

is executed on a single thread, redesigning a NIDS system is required to take advantages of the multi-core processor. We need a multi-thread NIDS.

When designing a multi-thread NIDS, reducing access synchronization on shared tables can dramatically increase the system performance since these operations cause threads stalled fpr process [3]. To solve this problem, some designs [4, 5, 6] allow each thread to maintain its own TCP reassembly table instead of sharing a global one; some others [7] uses a synchronization on TCP reassembly table. Comparatively, the former implementation strategies outperform the results of the latter ones.

Only one thread performing packet capture in current multi-thread NIDS implementations, packets are inspected in other threads [4, 5, 6]. Buffers are used between the packet capture and packet inspection threads. This simplfies the NIDS architectures, but data migration also introduces large cache misses.

A hash-based traffic splitting algorithm is usually used for distributing packets. This algorithm is proven to have an uneven workload distribution and hence Schuff et al. designed a new load balancing algorithm [6] for their implementation. Schuff's algorithm intends to balance workloads according to the number of packets waiting in the buffers. However, the number of buffered packets cannot represent the real workloads since the analysis time may vary from packets to packets. Besides, running a load balancing algorithm may consume some CPU cycles, hence load balance is not needed if CPU utilization is low.

1.2 Contribution

In this thesis, we propose a balanced multi-thread NIDS (bmtNIDS) to utilize the computing capacity of a quad-core system. All threads can perform packet capture simpultaneously, buffers

between a packet capture thread and other packet inspection threads are no more required. We leverage the packet filter in `libpcap` to build up a traffic splitter, which prevent threads from receiving duplicate packets by distributing traffic into flows. In addition, we propose a passive load balancing (PLB) algorithm to alleviate the workload on busy threads. PLB algorithm determines the workload of threads based on the CPU utilization since threads are bound one-on-one to specific cores. For optimization, PLB is activated only when the CPU utilization is higher than a pre-defined threshold.

1.3 Synopsis

The thesis is organized as follows. We review design principle of multi-thread NIDS and some parallelism for NIDS in Chapter 2 and Chapter 3. Then, we propose the balanced multi-thread NIDS (bmtNIDS) in Chapter 4. Chapter 5 describes the implementation of bmtNIDS based on Snort, and we show some experiments and the performance in Chapter 6. In the end, we conclude the thesis in Chapter 7.

Chapter 2

Design Principles

Vermerien and Haagdorens [3, 7] proposed three principles when designing a multi-thread NIDS:

- Order of Seniority Processing

  Order of seniority processing (OoSP) is required for processing packets in the receiving sequences. This is a very important constraint in most network applications since connection states could only be built up by a correct packet processing order. For example, NIDS relies on the seeing of a three-way handshake (SYN, SYN-ACK, ACK) to recognize the establishment of a TCP connection. Any alternations of this order would confuse the NIDS and cause alerts generated, NIDS may also miss attacks if the connection states dose not be properly recorded. Thus, one packet could be analyzed only when all packets before this packet are processed in a multi-thread NIDS.

- Workload balance

  A multi-thread NIDS may have a better performance than a single thread one since its workloads are shared by multiple computing cores. But this benefit could only be achieved under an even workload distribution. If one thread receives a larger packets which exceeds its process ability, this thread becomes the system bottleneck and NIDS starts to loss packets. Therefore, a load balancing algorithm is indispensable for a multi-thread NIDS.

- Access synchronization

  Shared tables may be accessed simultaneously in a multi-thread NIDS, a designer must guarantee these tables are thread safe to prevent the race condition. For example, the Stream4 preprocessor in Snort maintains a TCP reassembly table, out-of-order TCP segments are reassembled into ordered sequences before passing them to the detection engine [8]. But, such a TCP reassembly table should be safely protected on designing a multi-thread Snort since all threads would perform packets reassembling concurrently. Thus, synchronizations are required to prevent shared tables from data corruptions.

Chapter 3

Related Works

Conventional terminology classifies parallel computing into two categories: data parallelism and function parallelism [9]; data parallelism scatters data across multiple computing units which perform same calculations, while function parallelism divides the entire process into multiple sub-functions and alalyzes data on all sub-functions concurrently. Based on the conventional classification and the characteristic of NIDS, Wheeler et al. [10] claimed that NIDS could be paralized in three different levels, which are node level (NIDS cluster), component level (pipeline architecture), and sub-component level (multi-thread NIDS), all aforesaid methods could be implemented using either data parallelism or function parallelism.

In this chapter, we follow the classification proposed by Wheeler to review different parallelisms of NIDS and point out their pros and cons, which covers Section 3.1 to 3.3; we make a summary at the end of this chpater.

3.1 Node Level Parallelism

NIDS cluster usually focuses on high speed network, it combines multiple low-cost commodity hardwares with identical or different configurations to perform inspections on traffic. Each node inside NIDS cluster only needs to analyze a portion of network traffic, so packet loss could be reduced and deeper inspenctions are allowed. Moreover, the scalability of NIDS cluster could be extend by adding nodes to the cluster system, increase on traffic is allowed.

Kruegel et al. [1] used one scatterer and multiple slicers to split traffic into frames and route frames to different IDS sensor sets, each sensor analyzes a subset of frames according to its pre-defined attack scenarios. Le et al. [11] proposed Benefit-based Load Balancing algorithm and implemented it on a FPGA device, flow information and load-balancing are taken into concerns when distributing traffic, this guarantees packets corresoding to same attack would be analyzed in same node. Vallentine et al. leveraged the independent state framework of Bro [12, 13] to build up a communication protocol for NIDS cluster, correlated information and low-level analysis results are shared between nodes to reduce the miss of attacks [2].

3.2 Component Level Parallelism

The data flow of NIDS could be divided into several connected functional steps, executing each step by different thread or process could utiltize all comuting resource of a multi-core system. However, we can't divide NIDS process into equal workload steps, since some functions are not dividable. Antonatos et al.'s research showed that signature matching usually occupies more than 50% of packet process time [14], Schuff et al.'s evaluation also got the same conclusion: string matching would become the bottleneck in a full pipelined NIDS [6]. Buffers shared between functional steps also lower the performance, since buffer access needs be protected by synchronizations[7], and data migrations would introduce large cache misses [5].

3.3 Sub-component Level Parallelism

Since performing signature matching takes most computing resources, it becomes a good choice to parallelize the whole packet analysis process (including signature matching and other analysis functions) on a multi-core system. Traffic distribution are added before parallelized

analysis process like the front-end devices in a NIDS cluster, threads only need to face a subset of traffic, so workload of NIDS could be shared by multiple computing cores.

Yu et al. [15] divided the payload of packets into several equal length fragments and performed Aho-Corasick (AC) [16] signature matching algorithm on fragents by multiple threads concurrently. Since the time complexity of AC algorithm depeneds the length of the packet's payload, reducing the length of payload could increase the process speed. Each fragment also has a overlaps with previous one, this avoids the signatures occured at the edge of fragments.

Martin Rosech announced the next generation Snort, Snort 3.0 [17]. Differ form previous version, Snort 3.0 was broken into two piceces, the Snort Security Platform (SnortSP) and the Snort Detection Engine. The former provides the common functions for packet porocessing and utilizes a dispatcher to coordinate information flow between analysis components, the later performs inspections on multiple detection engines parallelized with multi-thread.

Paxson et al. proposed a multi-threaded event-based system and utilized a custom FPGA device for receiving/forwarding packets [18]. In this system, threads perform inspection on packets and generate events, events are categorized into multiple independent event queues and stimulate more analysis. They also allow threads to process events from separate queues concurrently, but events inside a single queue must be processed sequentially.

Intel's multi-thread NIDS is based on Snort, they dedicated one thread to receive packets and split traffic into flows using a hash function, the other threads were used to perform packet inspection on received packets [4, 5]. Since there is no order relationship between flows, packets could be processed in different threads without violating the OoSP requirement. Threads are bound one-on-one to specific cores, this avoids the cache misses caused by threads context switched between cores; they also allow each thread maintain a TCP reassembly table instead to share a global one, since packets from same flows are distributed to one thread, time-consuming

synchronizations are no more required.

Schuff et al. also divided Snort's packet analysis process into producer (packet capture) routine and conumer (packet inspection) routine [6], but unlike Intel's method, there is no dedicated thread for receiving packets. All threads could perform either packet capture or packet inspection, but only one thread is allowed to receive traffic at a time, other threads perform inspection on packets until the buffer inside the packet capture thread is full. Load balacing algorithm is also proposed to instead the hash-based distribution, new flows are assigned to the thread which has less buffered packets, since hash function has been proved to have a uneven workload distribution under real world conditions.

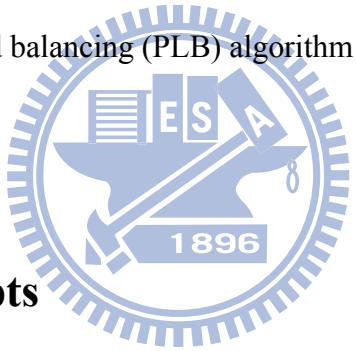
3.4 Summary

While designing a multi-thread NIDS on a multi-core system, previous researches showed that it is better to split traffic into flows to keep the OoSP requirement, this also allows threads to access a shared table without any synchronization. However, both Intel's and Schuff's multi-thread NIDS requires to transfer packets between packet capture thread and packet inspection threads, this generates large cache misses when data migration. Determining workloads depending on the number of buffered packets inside threads are still not precise, since packet process time is tightly depending on the characteristics of packets and detection rules defined by users. Load balancing function is not necessary to be performed all the time, since NIDS doesn't always suffer a high volume netwrok traffic. In this paper, our multi-thread NIDS is proposed to solve the aforesaid problems.

Chapter 4

Balanced Multi-thread NIDS

In this chapter, we propose a balanced multi-thread NIDS for multi-core systems (we abbreviate this as "bmtNIDS" in following paragraph). Section 4.1 outlines the design issues of our bmtNIDS. Section 4.2 depicts the multiple-packet-capture (MPC) architecture which allows all threads perform packet capture simutaneously without receiving any duplicate packets. Section 4.3 describes the principle for share tables division to reduce access synchronizations. Section 4.4 introduces the passive load balancing (PLB) algorithm into MPC architecture for balancing workloads among threads.



4.1 Design Concepts

We follow the following six design issues when implementing our multi-thread NIDS:

- Capture traffic simultaneously

  All threads are allowed to perform packet capture simultaneously without receiving any duplicate packets. This eliminate the data migrations between the dedicated packet capture thread and the packet inspection threads.

- Analyze flows

  Different flows could be processed in different threads without violating the OoSP requiremnet, since there is no order relationship between flows. Furthermore, splitting traffic into flows could be achieved by a simple hash function, this also becomes another

reason to choose flows as our minimum inspection unit.

- Reduce access synchronization

  Some shared tables could be divided into multiple sub-tables, each of them is maintained by one thread. Threads are allowed to access their own sub-tables without worrying about race condition, this minimizes the overhead on waiting access synchronizations.

- Bind threads to cores

  One thread is allowed to be executed on one specific core using the thread affinity scheduling provided by NPTL [19], since scheduling a thread from one core to another would generate large cache misses. The number of threads is better not to exceed the number of cores inside our system, otherwise, context switch would happened frequently and lower the locality of refernence.

- Monitor CPU utilizations

  The CPU cycles each thread spends on inspecting packets is directly proportional to the CPU utilizations of each core, if threads are bound one-on-one to cores. Monitoring CPU utilizations also brings two advantages: (1) predicting the available computing resources more precisely, since the process time is tightly depending on the characteristic of packets; (2) takeing interrupt services and other applications into concern, because some of them, especially network interrupt services, would take a lot of computing resources.

- Balance workload

  Strong load balancing algorithms are necessary for a multi-thread NIDS, however, load balance only need to be performed on threads which is apt to be overloaded. In a low network traffic evironment, even a few threads could handle all coming traffic, as long as the cores runs these thread are still have available computing resources.

4.2 Multiple-Packet-Capture Architecture

This section we introduce the multiple-packet-capture (MPC) architecture which allows threads performing packet capture simultaneously without any duplicate packets be received.

Unlike previous researches [4, 5, 6] that dedicated one thread for packet capture and use the remaining threads to inspect packets, we let all threads perform entire "packet process steps" as a traditional single-thread NIDS. All threads in our design could perform packet capture at the same time, buffers between packet capture thread and packet inspection threads are no more required. But performing packet capture simultaneously comes another question: how could we avoid packets being received twice?

4.2.1 Kernel Space Traffic Splitter

To solve this problem, we leverage the packet filter mechanism provided by libpcap [20, 21] to implement a traffic splitter. Most NIDSes support users to receive their interesting packets by setting a packet filter, which discards the unwanted packets in the krenel space. Based on this characteristic, we classify the rules of packet filters into two categories depending on its usage:

- Site-policy-related rules

  The site-policy-related rules filter out the pcakets with specific protocols, ip addresses or port numbers. For example, "$host$ 140.113.1.1" receives those packets which have 140.113.1.1 in the source or destination IP address, "$port$ 21" considers those FTP traffic.

- Traffic-splitting rules

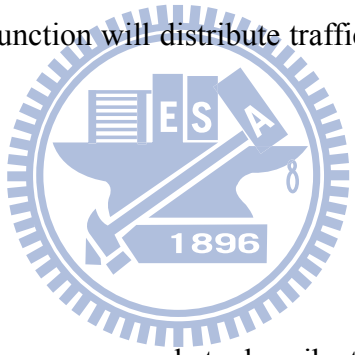
  The traffic-splitting rules are the hash functions for distributing packets. Libpcap supports more complex expressions in filter rules, such as arithmetic operators $(+, -, *, /)$ or binary operatiors $(>, <, =, !=)$. We use these operators to implement a hash function. For

example:

$$((ip[12:4] + ip[16:4])\ \&\ 0x01) == 0$$

where $ip[12:4]$ and $ip[16:4]$ indicate the source IP address and the destination IP address. Port numbers are not included in the hash functions, since some protocols, like ICMP messages, don't have a port number in their headers. A complicated hash function also increases the process time spent on matching packets with filter rules.

All threads have the same site-policy-related rules, but the traffic-splitting rules are different with each other at the right of equal statement for distributing packets. Thus, threads in MPC architecture could perform packet capture simultaneously, and no packets will be received twice. Moreover, since hash function will distribute traffic into flows, the OoSP requirements are satisfied.



4.2.2 Case Study

We use a quad-core system as an example to describe the packet process inside MPC architecture. Our multi-thread NIDS executes 4 threads, $thread_0$ ~ $thread_3$, all threads are designed to inspect the traffic in/out a web server cluster. For concurrent packet capture, $Thread_N$ compiles the following rules :

$$(((ip[12:4] + ip[16:4])\ \&\ 0x03) == N)\ \&\ ((port\ 80)\ ||\ (port\ 443))$$

where $port\ 80$ is the default HTTP server port number, $port\ 443$ receives the HTTPS traffic, and N is equal to 0 ~ 3. $Thread_N$ only receives those HTTP or HTTPS packets which have a hash result equals to N, but all packets are guaranteed to be inspected by one of the four threads. We also bind threads one-on-one to cores using thread affinity function, this avoid large cache misses caused by scheduling thread from one core to another.

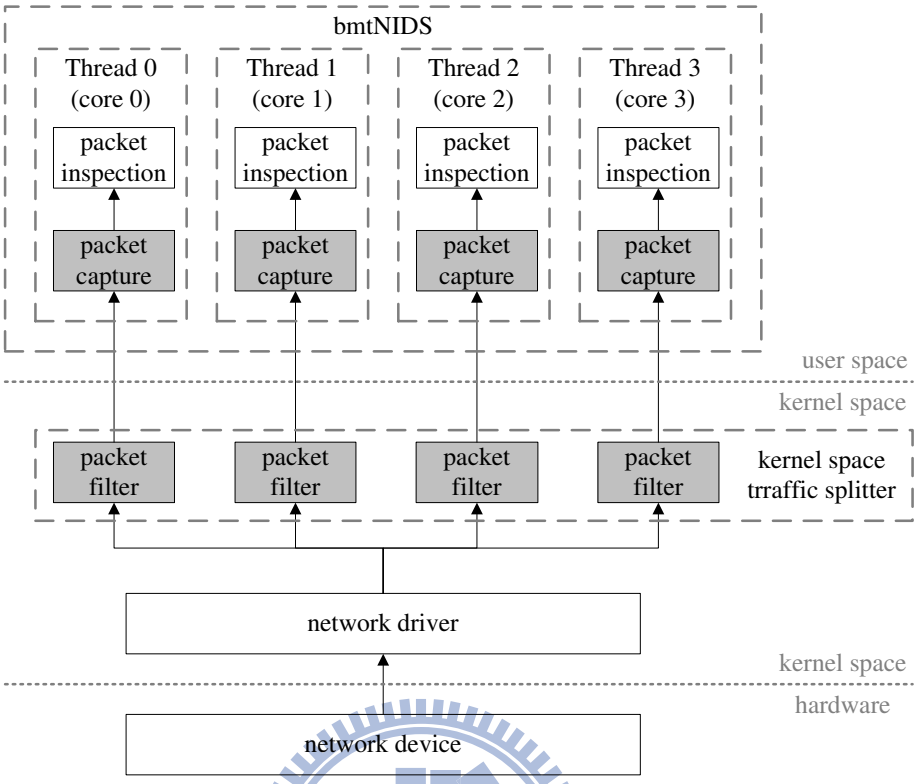


Figure 4.1: bmtNIDS architecture

When network device receives packets, it sends an interrupt to the kernel to request a service. After network interrupt service routine move packets from the buffer on network device into memory, packets are soon passed to the packet filters installed by threads. If all rules compiled by one thread are matched, packets would be copied into the socket read buffer of this thread [22, 23]. (Socket read buffers is used in Linux system, for BSD, packets are copied into the STORE buffer of BPF [24].) Dropping packets is only occured at failed matching on site-policy-related rules, otherwise, packets would be copied into one of the socket read buffers depending on the traffic-splitting rules. Finally, packets are copied into user space by a $read()$ system call, and threads start to perform inspections on these packets. The flow of packet process are shown in Figure. 4.1.

4.3 Access Synchronization

Access synchronizations are necessary for a multi-thread NIDS to protect shared tables from race conditions, but this increases the time threads stalled for waiting execution. Allowing threads has seprate tables could remove the requirements of synchronizations, but some detections, such as port sweep detection, still need the cooperation of all threads. To find a balance between these two extreme cases, we classify shared tables into flow-based tables and network-based tables.

Flow-based tables save the information about indiviual flows, such as connection statuses of TCP traffic. Since packets from the same flow are sent to the same thread, the table contents about this flow would not be modified by more than two threads. Thus, one flow-based table can be divided into multiple sub-tables, and each thread is allowed to access one of the sub-tables without any synchronization.

On the other hand, network-based tables concern the traffic of the whole network, relationships between flows are recorded. Any two packets from different threads may update the same location inside a network-based table, so these tables should be shared in a multi-thread NIDS. Thus, synchronizations are required to protect network-based tables from race condition.

Based on this classification, dividing flow-based tables would not affect any detection accuracy, since inspections on entire network still record their detection result in one network-based table. Moreover, we use spin locks to implement the access synchronizations rather than mutex locks, since most critical sections only takes a little CPU cycles, busy waiting on locks is faster than stalling thread into blocked states. Thus, our system could have a minimum overhead on dealing with the access synchronizations.

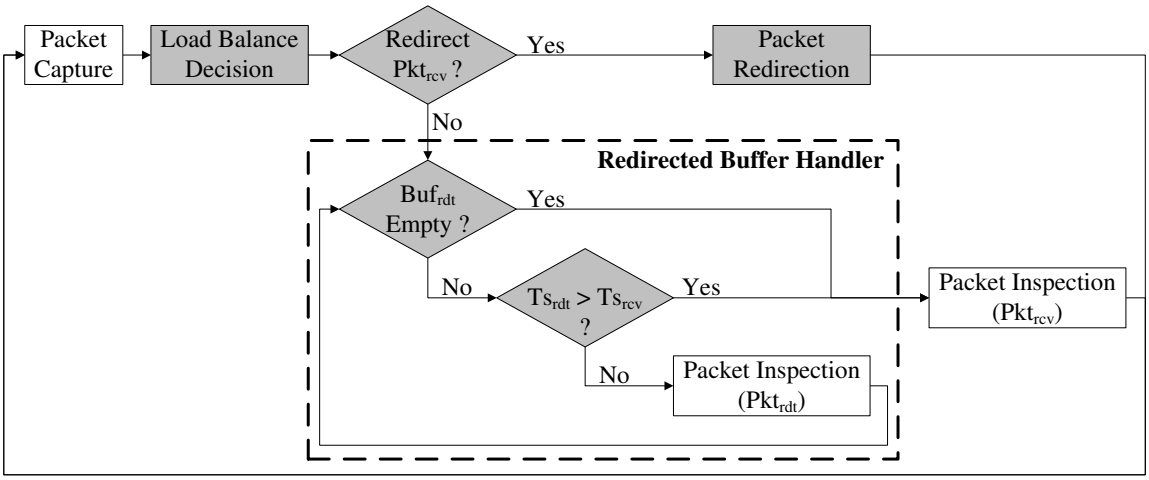


Figure 4.2: The data flow of thread with passive load balancing algorithm

4.4 Passive Load Balancing Algorithm

Although hash function guarantees a even output in theory, it was proven to have a nonuniform traffic distribution under realistic enviroment [6]. Since workloads of threads tightly depend on the traffic distribution, strong load balancing algorithm are required to let newcoming packets be analyzed in threads which have available computing resources.

Balancing workloads aggressively is necessary for a multi-thread NIDS which only dedicates one thread for receiving packets, packet capture thread would distribute packets to other packet inspection threads; but for our MPC architecture, only those threads which tend to be overloaded should enable a load balance function, since all threads could receive packets by themselves. We call this workload balance algorithm as "passive load balancing" (PLB) algorithm.

Figure 4.2 shows the modified thread with PLB algorithm. Load balance decision is performed when thread receives a new packet, Pkt_{rcv}, this packet will either be analyzed in local thread or be balanced to another thread depending on the workloads. Meanwhile, packets from other threads (we call these packets as $Pkt_{rdt}s$) will also be handled in the original packet in-

spection routine. We depict the data flow in the following paragraph.

4.4.1 Load Balance Decision

We first define the meaning of overwhelmed thread and non-overwhelmed thread. If one thread's workload is larger than a predefined threshold τ, we call this thread as a overwhelmed thread, otherwise, it is a non-overwhelmed thread. Overwhelmed threads usually result from a suddenly coming traffic or some process time consuming packets, these make threads start to loss packets.

When one thread becomes a overwhelmed thread, we start a packet redirection mechanism to redirect new coming packets to a non-overwhelmed thread. A overwhelmed thread doesn't spend cycles on performing time-consuming inspections on received packets, it redirects packets to a non-overwhelmed threads and starts to receive next packet; non-overwhelmed thread will buffer redirected packets and take over the sucessional process on them. This avails a overwhelmed thread receive more packets which could be dropped due to socket read buffer full. But if there is no non-overwhelmed thread exist, i.e. all threads are busy in packet processing, received packets are still analyzed in local threads until some threads are avaiable to process more packets.

Moreover, a overwhelmed thread may still get packets from other threads, while a non-overwhelmed thread may also redirect packets to others, since we must satisfy the OoSP requirement. According to our second design issue, packets from same flow should be analyzed in same thread, only new coming flows could be redirected to non-overwhelmed threads, old flows should be analyzed in local thread. Packet-to-thread relationship is also recorded in a "flow table", so threads could look up flow table to decide which threads should analyze these packets. Algorithm 1 summarize the process of passive load balance decision.

Algorithm 1 The process of load balance decision

```
1:  if Pkt_rcv is not the first packet of a flow then
2:      N = lookupFlowTable(Pkt_rcv)
3:      if N is not local thread then
4:          redirect Pkt_rcv to thread N
5:          receive next packet
6:      else
7:          analyze Pkt_rcv in local thread
8:      end if
9:  else if local thread is a overwhelmed thread and a non-overwhelmed thread M exist then
10:     record thread M into flow table
11:     redirect Pkt_rcv to thread M
12:     receive next packet
13: else
14:     analyze Pkt_rcv in local thread
15: end if
```

4.4.2 Redireted Buffer Handler

All threads implement a redirected buffer (BUF_{rdt}) to store packets from overwhelmed threads, we also modify the original packet inspetion routine to process these packets. Moreover, we claim that all redirected packets come before Pkt_{rcv} should be handled before Pkt_{rcv} be analyzed.

Before thread performs inspections on Pkt_{rcv}, it checks its BUF_{rdt} first. If BUF_{rdt} is empty, thread continues to analyze Pkt_{rcv}; but if BUF_{rdt} is not empty and its first packet, Pkt_{rdt}, has a timestamp Ts_{rdt} smaller than the timestamp Ts_{rcv} of Pkt_{rcv}, thread performs inspections on Pkt_{rdt} and goes back to check next redirected packet in BUF_{rdt}. Pkt_{rcv} will not be processed until BUF_{rdt} is empty or Pkt_{rcv} is the latest packet. The strategy of packets processing order is detailed in Algorithm 2.

4.4.3 Workload determination

At the end of this section, we describe how we determine the workloads. The workloads of threads is decided by how many cycles one thread spends to perform inspections on the packets

Algorithm 2 The process inside redirected buffer handler

```
1: while BUF_rdt is not empty do
2:     Pkt_rdt = first packet in redirected buffer
3:     if Ts_rdt < Ts_rcv then
4:         packetInspection(Pkt_rdt)
5:     else
6:         break loop
7:     end if
8: end while
```

it received in some periods of time (including the cycles threads stalled for memory access). If the cycles one thread should take exceeds the processing ability of the CPU, i.e. packets received could not be processed in that period of time, we say this thread is overlading and start to loss packets.

However, profiling the CPU cycles one thread has spent is hard and resource wasteful, counting how many waiting packets inside each thread also could not correctly present the work-loads, since the process time of packets depends on the characteristics of traffic and detection rules defined by users. In this paper, we choose to monitor the CPU utilization instead.

Remember in 4.2.2, we bind threads one-on-one to specific cores. This not only prevents threads from scheduling one core to another, but also enables us to monitoring the resource consumption of each thread, since the cycles one thread has spent is directly proportional to the CPU utilization of the core executing this thread. Monitoring CPU utilizations also takes interrupt services and other process into concern, this allow us to predict the available computing resource more precisely rather than couting waiting packets.

The CPU utilization is monitored by one isolated thread (we call this thread as monitor thread), monitor thread shares a set of flags with other threads which run the packet inspection. If the CPU utilzation on one core is higher than the threshold τ, monitor thread sets the flags to notify the thread executing on this core becomes a overwhelmed thread, load balance decision also depends on these flags to decide whether new coming flows should be redirected and which

thread could inspect more packets.

Chapter 5

Implementation

In this chapter, we show how we implement our bmtNIDS based on Snort (we abbreviate this as "bmtSnort" in following paragraph). We first describe the Snort NIDS in Section 5.1, Section 5.2 details the modification on Snort to realize our balanced multi-thread NIDS.

5.1 Snort

Snort [8] is one of the most popular NIDSes among all currently availble selections, since both the software itself and its detection rulesets are freely available, open sourced, and maintained by groups of development communities. Snort is primarily a signature-based NIDS, it performs string or regular-expression matching on the packets' headers and payloads. The analysis process of Snort is composed of following five stages, the data flow of Snort is also shown in Figure 5.1:

- Packet capture

  Snort uses the platform independent packet sniffing library, libpcap [20], to read network traffic. Libpcap sniffs packets on datalink layer and retrieves raw packets into user space

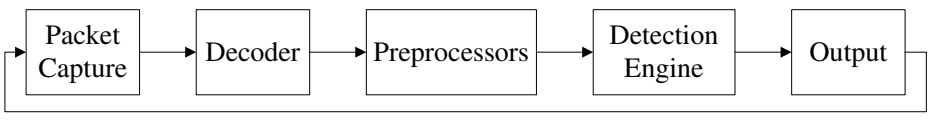


Figure 5.1: The data flow of single thread Snort

application. Raw packets contain all protocol information required by Snort to perform analysis on packet's header and record the state of connections.

- Decoder

  After packets are captured into application, Snort first decodes the protocol of each packet in this stage. Snort seriatim decodes the datalink layer, network layer, and transport layer protocols, then it saves decoded information into a data structure and moves packets to preprocessor stage.

- Preprocesors

  The usage of preprocessors falls into two categories: packet normalization and anomaly-based detection [25]. The former normalizes packet's header and payload to avoid insertion or evasion attacks, while the later focuses on the malicious which correlates with multiple hosts or doesn't have a discernable signature. Preprocesors stage is implemented in plug-in architecture, this avails users to write their own preprocessors; Snort also provides some basic preprocessors as default settings, such as Flow, Frag2, Stream4, HTTP_decode, and Portscan2.

- Detection engine

  Normalized packets are sent to this stage. Detection engine has two major works: rule parsing and signature detection. Rule parsing parses detection rules definded by user to build up attack signatures at initialiozation, Snort uses these attack signatres to perform string matching algorithms, such as Aho-Coarsick [16] or Wu-Manber [26], on packets' headers and payloads.

- Output stage

  Both preprocesors and detection engine may generate events. Snort logs alerts in various

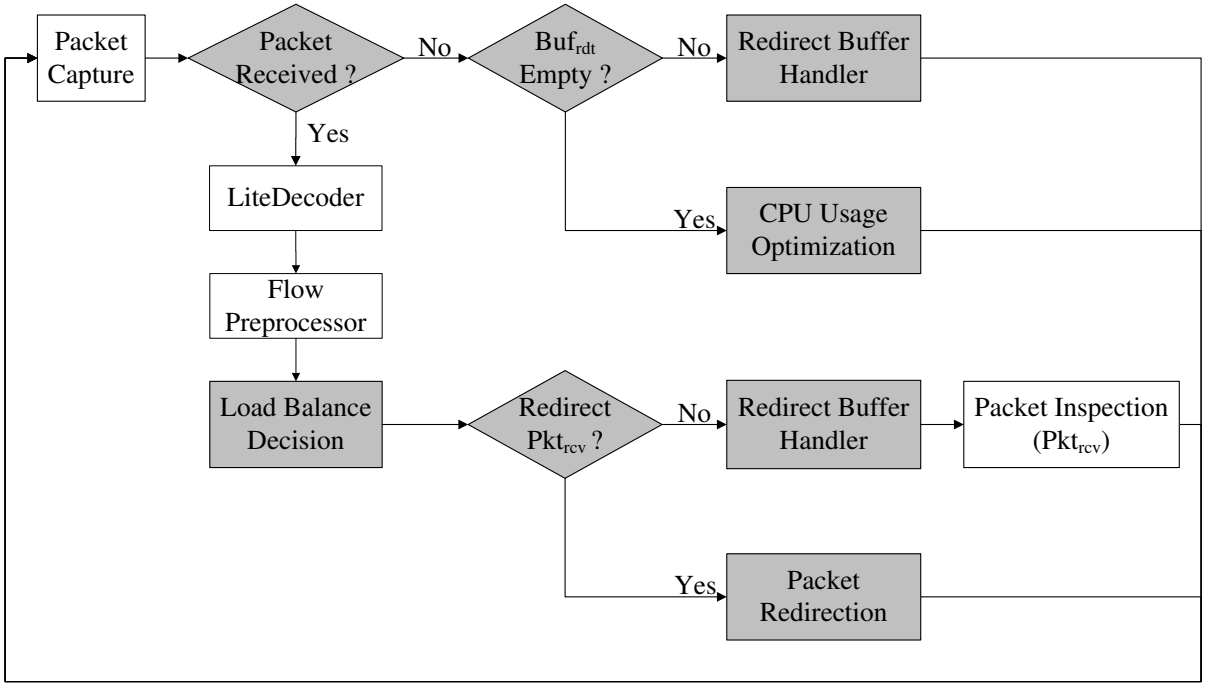


Figure 5.2: The data flow of MPCPLB-Snort

ways, such as text files, syslog, or saving alerts in a relational databases; it could also save data in differnt formats, such as XML. Snort defined binary format is also a common choice[27].

5.2 Balanced Multi-thread Snort

The data flow of bmtSnort is shown in Figure 5.2. We use a non-block receiving function to capture traffic from network, threads will perform different works depending on whether packet is received. When thread gets packets, packets would either be redirected to other threads or be analyzed in local based on the result of load balance decision, redirected packets are also handled before Pkt_{rcv} be inspected. Even receiving function returns without any packets, thread still performs inspections on packets inside BUF_{rdt} unless buffer is empty. Only if no packet is received and BUF_{rdt} is empty, we execute the CPU usage optimization to eliminate the busy

waiting resulting from non-block packet capture.

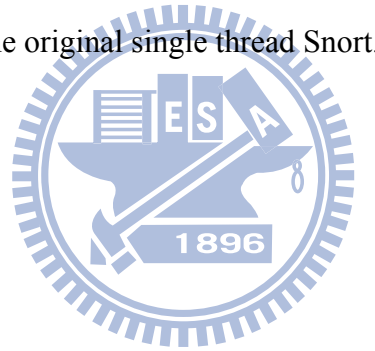
To implement the flow table for packet-to-thread lookup inside load balance decision, we leverage the design of multi-thread NIDS proposed by Schuff et al. [6]. Flow preprocessor is moved outside the original packet process of Snort, since its packet-to-flow searching performs the same work as packet-to-thread lookup. They also implemented a simple packet decoder (which are called a LiteDecoder) to retrieve the information (i.e. ip address and port numbers) required for Flow preprocessor. If Flow preprocessor shows Pkt_{rcv} is the first packet from a new flow, load balance decision will choose one thread to analyze all packets from this flow according to threads' workloads; otherwise, thread bypasses the load balance decision, either redirects or performs inspections on Pkt_{rcv} depending on lookup result.

Packets inside BUF_{rdt} would be analyzed in two cases, one is before Pkt_{rcv} be inspected, another occured at no packet received but BUF_{rdt} is not empty. If thread receives packets, we use the timestamp of received packets (Ts_{rcv}) as the comparing reference, which we proposed in Section 4.4.2. In second case, no Ts_{rcv} can be referneced, we record current time, Ts_{curs}, as the timestamp to decide when to stop the redirected buffer handler instead. Using a non-block receiving function and handling redirected buffer when no packet received are necessary, since a buffer overflow may occured if one thread continously gets redirected packets from other threads without any packets received by itself.

CPU usage optimization is realized by a sleep function and a set of counters maintained by each thread, we use these counters to track network activity. If threads receive traffic or handle packets inside BUF_{rdt}, counters are increased; otherwise, we decrease counters by one. Thus, a larger counter value indicates threads are busy in packet processing, while low network activity would have decresing counters. When counters decreased to zero, we think there is no packets arrived at this time and force thread go to sleep to relinquish the processor voluntarily. This

not only prevents threads from looping in the packet capture stage, the available CPU cycles also allow other process inside the system to be executed. Counters are reseted to a non-zero value after thread regains the CPU control, so thread will not sleep a long period of time and could repsond to new coming packets more quickly. Another reason for optimizing CPU usage is that: our bmtNIDS determines threads' workloads depending on the CPU utilizations, but a busy waiting thread will consume all computing resources so the PLB algorithm becomes useless.

Finally, both Pkt_{rcv} and Pkt_{rdt} would be analyzed as the flow in Figure 5.1, threads go through decoder, preprocessors (Flow preprocessor is excluded, of course), detection engine, and output stage to perform a complete inspections on packets and log alerts. Thus, our bmtSnort could perform detections as the original single thread Snort.



Chapter 6

Experiments

In this chpater, we test the performance of our bmtSnort. We first describe the expriment environment and system setup, then several experiments are launched to show the performance improvements from bmtSnort. We make a summary at the end of this chapter.

6.1 Experiment Setup

To test the performance of NIDS with real network traffic, we use a traffic generator to resend recorded packets to a NIDS. The experiment setup is shown in Figure 6.1. NIDSes are run on a machine using one 2.40GHz quad-core Intel Q6600 processor, it also contains two 4 MB L2 caches shared between cores and 2GB RAM as system memory. Traffic generator has one 3.2 GHz Intel Celeron D CPU with 512KB L2 cache and 1GB system memory. NIDS machine and traffic generator are directly connected by a gigabit link to eliminate the traffic from other host, INTEL 9301CT network adapter are equipped on both side to achieve gigabit transmission/ receiving rate. Both systems runs under Linux kernel version 2.6.24, the GNU C library 2.3.7 with Native Posix Thread Library are also installed to support thread affinity scheduling.

For a controllable traffic generation, we use Tcpreplay to resend pre-captured packets to NIDS. Tcpreplay [28] provides a reliable and repeatable mean to test network devices and applications, it reads network traffic from files saved by tcpdump and replays traffic on the network, packets are also allowed to be transmitted as the recorded speed or specified data rate.

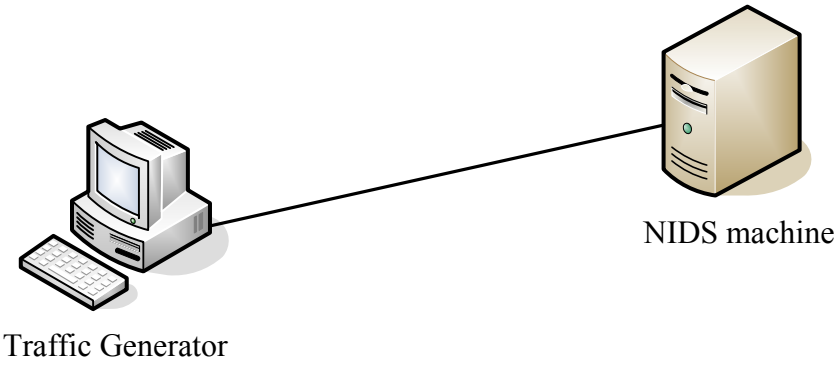


Figure 6.1: Experiment setup

Single thread Snort and Schuff's multi-thread Snort [6] are choosed as the comparison with our bmtSnort, all of them are well configured to run on the NIDS machine. Because both bmtSnort and Schuff's implementation are based on Snort version 2.6RC1, we also download this version from Snort website although it is not the latest release. Also, since Schuff's achievement focused on offline traffic inspection, we slightly modified their code for reading live network traffic. But this brings two different implementation depending on whether threads would be blocked on packet receive function, $pcap\_loop()$ or $pcap\_dispatch()$ are used for reading packets in block or nonblock mode (we also abbreviate them as block-Schuff and nonblock-Schuff). When nonblock-Schuff returns from packet recevie function without any packet, it calls this function again immediately to check if there is any packets can read. In this chapter, both cases will be discussed.

The detection rules for preprocessors and detection engine are downloaded from Snort website, all rules are enabled but those deleted or deprecated. Rule variables which should be configured by user are set to ``any", such as ``home net" or ``HTTP servers", this allows maximum intrusions be detected. Snort 2.6RC1 supports both Aho-Coarsick and Wu-Manber string matching algorithms for detction engine, we choose Wu-Manber to perform packet inspection.

The packet trace used to test the system performance comes from ``Defcon 9 Capture the

Flag contest", a system attack and defence contest between hackers held at July 2001. This trace (we abbreviate it as DEFCON trace) contains 3591523 packets with 665Mb and generates more than 10,000 alerts by Snort offline detection, this would be a pathological case for NIDS since huge amount of attacks and anomalous traffic are logged.

6.2 Results and Analysis

In this section, we test the performance of our bmtSnort and show the experiment results. First, we test the impact of PLB algorithm to demonstrate the requirement of a load balance function; second, we shows that a complicated packet filter would not have large effect on the system performance; finally, bmtSnort is compared with single thread Snort and Schuff's multi-thread Snort to show our improvements.

Tcpreplay is operated under different transmission rate to test the performance of a NIDS under different network environment. Although we can specify the desired transmission rates as input parameters, real outputs will be a little larger than expected.

How many percent of packets could be inspected will be compared (we call this as ``packet inspection ratio"). Only packets which are fully inspected are counted, our concern is that packet loss should be considered as a miss even the same number of alerts are generated.

6.2.1 Benefits from PLB Algorithm

To test how PLB algorithm improves the packet inspection ratio. We disable the load balance decision and monitor thread of bmtSnort, all received packets are analyzed in local thread. We run 4 threads in both settings, PLB algorithmson a balanced bmtSnort are launched when CPU utilization exceeds 85%.

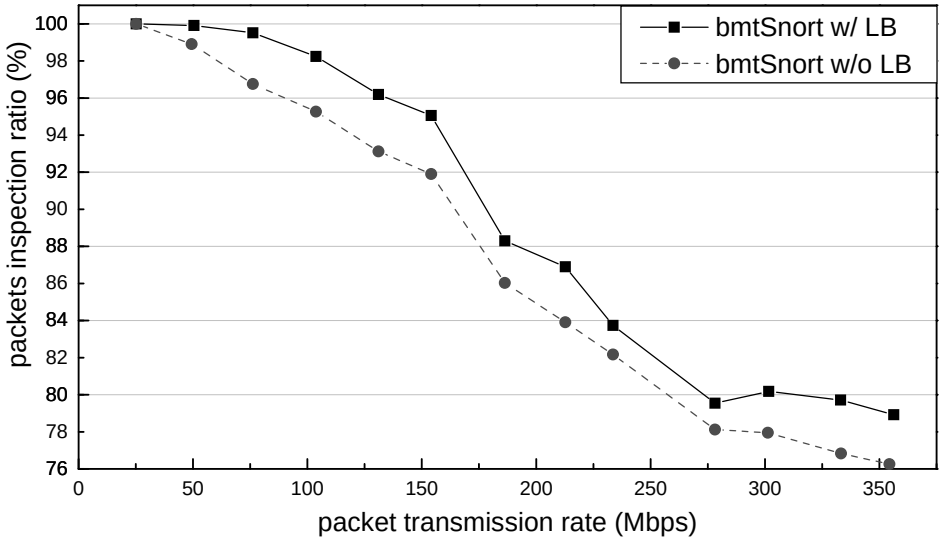


Figure 6.2: Performance improvements by PLB algorithm

The experiment result is shown in Figure 6.2. If transmission rate is lower than 50Mbps, both bmtSnorts can inspect more than 98% of packets. But with the raise of the network speed, the bmtSnort without load balance drops more packets than a load balanced one. This experiment shows that our PLB algorothm improves the packet inspection ratio by 2.5% in average, and this benefit is promoted when the transmission rates are increased.

6.2.2 Process delay in Packet Filter

To demonstrate packet filter would not have large effect on the performance, different site-policy-related rules are compiled to defer the packet filtering process. The compiled rules are shown as follows:

- Rule 1:

  $(tcp||udp||icmp)$

- Rule 2:

  $(tcpu||udp||icmp)$ && !$(host\ 140.113.1.1)$ && !$(host\ 140.113.40.36)$ && !$(host\ 140.113.6.2)$

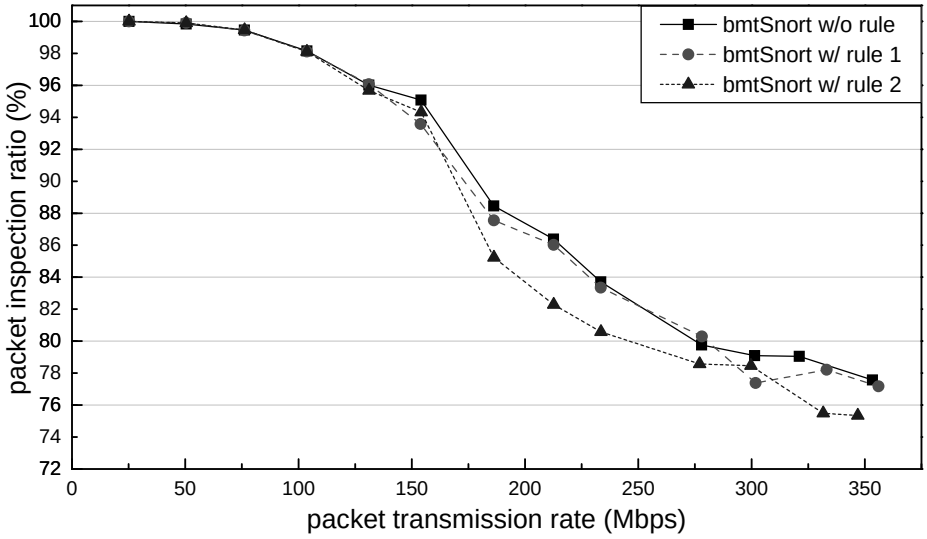


Figure 6.3: Performance effect of complicated packet filter

All site-policy-related rules are compiled with traffic-splitting rules at runtime, and a bmt-Snort with only traffic-splitting rules is used as the compared reference. We run 4 threads in bmtSnort with different seettings, PLB algorithms are launched when CPU utilization exceeds 85%. Input traffic which will be filtered out by rule 1 or rule 2 is also removed, so same number of packets is sent from traffic generator.

The comparisons are shown in Figure 6.3. If the transmission rate is less than 150 Mbps, the similar performances are achieved. When we increase the packet transmission rate, bmtSnort with rule 1 can still follow the trend of bmtSnort without rule. But when the transmission rate is larger than 154.14 Mbps, bmtSnort with rule 2 has an obvious performance down, it has a packet inspection ratio that is 2.5% lesser than the others.

Complicated rules amy defer the packet filtering in soft IRQ. In the worst case, one packet is compared with all filter rules but dropped finally. But in real situation, we seldom use address flitering. This restriction can be instead by setting the detection rules as ``pass'' on these host. The packet filter is usually used to filter specific ports or network protocols, so NIDS only

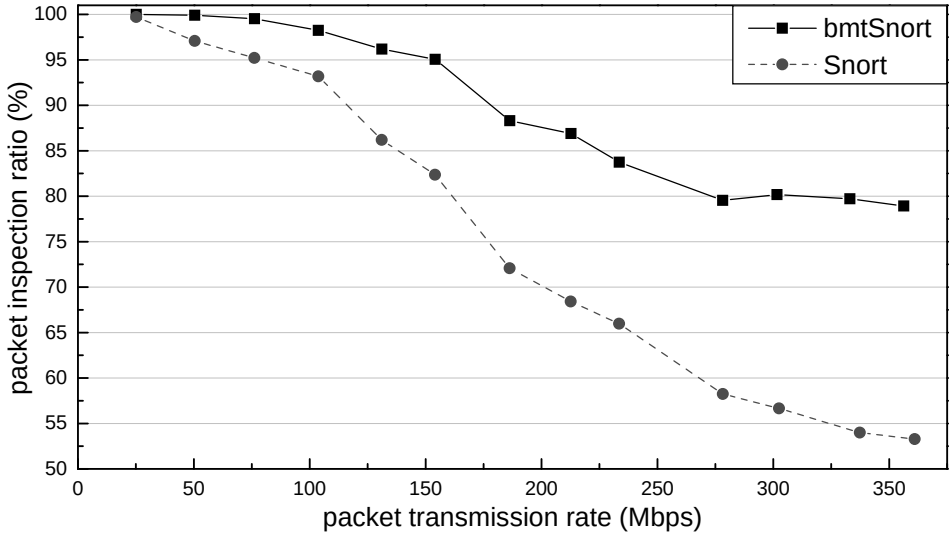


Figure 6.4: Performance improvements on Snort

needs to focus on a small portion of traffic. Thus, even complicated filtering rules are used, the performance of bmtSnort would not have large effect.

6.2.3 Improvements on Snort

In this experiment, we shows the improvements from bmtSnort. We run 4 threads in bmtSnort and intend to better utilize the computing capacity of a quad-core system. We also launch the PLB algorithm when CPU utilization exceeds 85%.

The experiment results are shown in Figure 6.4:

- Snort

  When the transmission rate is higher than 100 Mbps, Snort loses more than 5% of packets. With the raise of network speed, the packet inspection ratio decreases obviously. When the transmission rate is higher than 250 Mbps, less than 60% of packets can be inspected by a single thread Snort.

- bmtSnort

  With the transmission rate lesser than 154.14 Mbps, bmtSnort can analyze more than 95% of packets. Even the network speed exceeds 300Mbps, more than 78% of packets are still inspected.

The performance of Snort is not improved by a factor of 4 expected, only 1.5 times of packets are inspected. Two possible reasons are: (1) Numerous alerts are genreated by DEFCON trace, since it cnotains lots of attacks. bmtSnort performs access synchronization on output stage, threads may spin on the lock for log permission. (2) Althougth threads are executed on different cores, same memory spaces are shared, memory contention would be occured frequently when threads perform inspections on packets. Both cases causes threads stalled for data access. Moreover, Schuff's experiment, also executed 4 threads concurently, showed that the offline inspection on DEFCON trace can only be improved by a factor of 1.6. Thus, even our bmtSnort can not reach the expected performance, it still has a 25% improvement on packet inspection ratio in high speed networks, and its performance is approximately to the result of offline inspection.

6.2.4 Comparison

Finally, we compare the performance improvements among bmtSnort and two different implementaions on Schuff's multi-thread Snort. Snort is used as a compared reference in this experiment. All multi-thread NIDSes are executed with 4 threads to maximize the effectiveness of a quad-core system, and the CPU utilization threshold for PLB algorithm is set to 85%.

The experiment results are shown in Figure 6.5:

- block-Schuff

  At the network speed lesser than 103.71 Mbps, it analyzes more than 94% of packets.

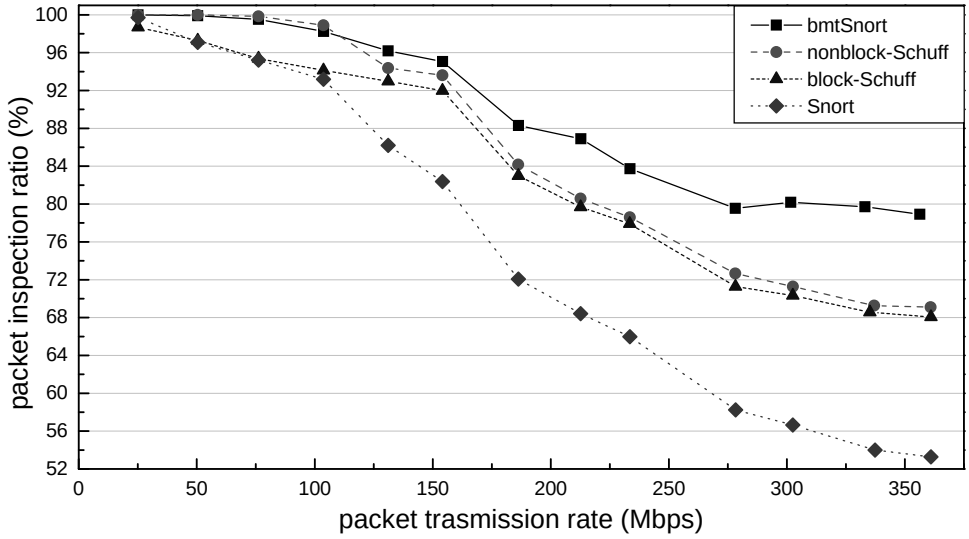


Figure 6.5: Performance comparison between bmtNIDS, nonblock-Schuff, block-Schuff and Snort

When the transmission rate is higher than 200 Mbps, less than 80% of packets can be inspected. Block-Schuff has a better packet inspection ratio than single thread Snort, but it starts to lose packets at a relative low network speed. block-Scufff can not reach the same performance as bmtSnort and nonblock-Schuff.

The block-Schuff sould have a poor improvements, since we use a block packet receive function. If there is no packet can be received, threads yield processor and enter to the blocked state. When the states change from blocked to running, threads will suffer from a delay due to context switch. In a low network speed environment, this context switch occurs frequently.

- nonblock-Schuff

  Even the trasmission rate raised to 103.71 Mbps, nonblock-Schuff can still analyze more than 98% of packets, it has a beter performance than block-schuff at low network speed. But with a higher trasmission rate, the packet inspection ratio only has a 1.1% advancement compared with nonblock-Schuff. nonblock-Schuff doesn't have a great improvement

in high speed network.

A nonblock receive function will force threads loop in the packet capture stage until they get packets, this reduces the time on waiting scheduling and context switch. But in a high network speed environment, NIDSes are rarely idle. This is why nonblock-Schuff has a better performance at low network speed, but can't achieve higher performance with the raise of transmissin rate.

- bmtSnort

  With the transmission rate lesser than 154.14 Mbps, bmtSnort can analyze more than 95% of packets. Even bmtSnort is operated at high network speed which exceeds 300Mbps, more than 78% of packets are still analyzed. But the experiment results also show that: With the transmission rate between 75 mbps to 125 mbps, bmtSnort has a smaller packet inspection ratio than nonblock-schuff, which are 99.52% compared with 99.84% at 76.17 Mbps and 98.24% compare with 98.9% at 103.71 Mbps.

  Two reasons make our bmtSnort analyze more packets in high speeed netowrk: (1) Since all threads perform packet capture simultaneously, more packets can be buffered in the kernel. Each thread has one socket receive buffer, so bmtNIDS uses 4 kernel buffers on saving received packets. (2) The load balance decision redirects packets according to the CPU utilization, this avoids threads getting more packets when they are suffering from high workloads. We have a better improvement under high transmission rate.

  The special case at low transmission rate may result following reasons: (1) We do not have a fine-grained sleep funtion for CPU usage optimization. Although the sleep function we used supports nano second of accuracy, a suddenly coming traffic may cause the socket receive buffer overflow while threads still block in the sleep function. On the contrary, nonblock-Schuff checks the scoket recevie buffer aggressively, the packet capture thread

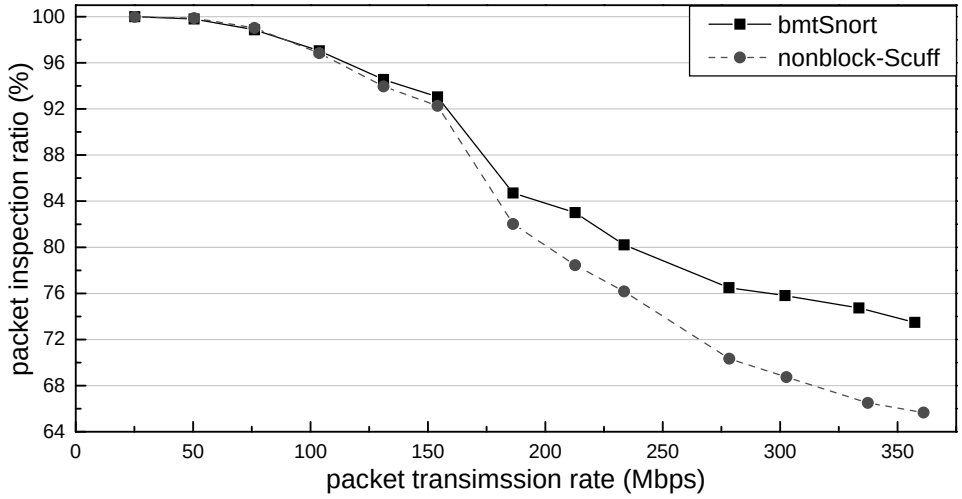


Figure 6.6: Performance effect of gnome system monitor

loops for new packets without performing ant inspections. This is why nonblock-Schuff has a better performance under low network usage.

However, nonblock-Schuff has a 100% CPU utilization, it forces thread busy waiting for new packets. If a computing-intensive application exists, nonblock-Schuff may compete the processors with this application. On the contrary, bmtSnort doesn't exhaust all computing resources, there are still idle cycles for other applications.

6.2.5 Performance down by gnome system monitor

In this experiment, a computing-intensive apllication is executed to affect the performance of bmtNIDS and nonblock-Schuff. We execute a gnome system monitor to generate a static resource consumption. Gnome system monitor reports system status periodically in a user specified interval, it consumes more CPU cycles at higher the monitoring frequency [29]. In this experiment, we set the gnome system monitor to report system status twice per second, which consumes about 17% of CPU resource (totally 400% in a quad-core processor).

The experiment result is shown in Figure 6.6. Both bmtSnort and nonblock-Schuff have a

performance down under high transmission rate, since gnome system monitor competes CPU resources with NIDSes. The performance of nonblock-Schuff is decreased at low transmission rate, which drops to 99.03% at 76.17 Mbps and 96.84% at 103.73 Mbps. On the other hand, gnome system monitor has little effect on bmtSnort, bmtSnort still maintains its packet inspection ratio, which is 98.86% at 76.17 Mbps and 97.04% at 103.73 Mbps. Gnome system monitor has more effects on nonblock-Schuff rather than bmtSnort.

Such applications which consume neglectable computing resources exist in real NIDS deployments. For example, a database or Barnyard. A database may be executed with NIDS on the same host for logging alerts, while Barnyard is applied to send the analysis results to a remote database for alerts aggregation [30]. Both cases may take an unpredictable CPU cycles, so NIDS can not waste all computing resources on looping for new packets. bmtSnort utilizes the CPU usage optimization to eliminate the busy waiting, it provides a good resource utilization strategy to share computing resource with other applications.

6.3 Summary

The PLB algorithm allows bmtSnort to have a 2.5% impact on packet inspection ratio. Even four thread executed concurretly, complicated packet filter rules doesn't have large effect on the performance. When performing inspections on DEFCON trace, bmtSnort improves the performance of single thread Snort by a factor of 1.5 at high network speed. Comparing with Schuff's multi-thread NIDS, we have a better performance in real netowrk environment: (1) bmtSnort has a packet inspection ratio 10% higher than nonblock-Snort at high transmission rate. (2) bmtSnort has a better strategy on resource utilization to coordinate with other applications.

Chapter 7

Conclusion and Future Work

In this thesis, we designed and implemented a new balanced multi-thread NIDS system, bmtNIDS, to get a better effeciecncy when running in a multi-core system. bmtNIDS supports threads performing packet capture concurrently without any duplicated packets. Flows are distributed by a kernel packet filter, so access synchronizations can be reduced. Since the flow-based tables can be devided to subtables and maintained by each thread. In our design, a PLB algorithm is also proposed to balance workloads by CPU utilizations, instead of counting buffered packets. The PLB algorithm redirects packets to non-overwhelmed threads only when CPU utilization exceeds the pre-defined threshold. We realize bmtNIDS on Snort and name the new implement as bmtSnort. By conducting a series of experiments with bmtSnort, single-thread Snort and Schuff's NIDS implementation, we compare and analyze packet inspectation ratios in terms of network speeds. The experiment results show that: (1) with higher transmission rate ($> 300Mbps$), bmtSnort improves the performance of Snort by a factor of 1.5. (2) comparing with Schuff's NIDS system, bmtSnort analyzes more than 10% of packets in a high speed network. (3) bmtSnort also supports a better strategy for sharing computing resources and hence the performance of bmtNIDS is not affected if the system also runs a computing-intensive application.

References

[1] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, "Stateful intrusion detection for high-speed network's," *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pp. 285--293, 2002.

[2] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The nids cluster: Scalable, stateful network intrusion detection on commodity hardware," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4637, p. 107, 2007.

[3] T. Vermeiren, E. Borghs, and B. Haaodorens, "Evaluation of software techniques for parallel packet processing on multi-core processors," in *Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE*, 2004, pp. 645--647.

[4] Intel, "Supra-linear packet processing performance with intel multi-core processors," 2006.

[5] E. Verplanke, "Understand packet processing with multi-core processors," 2007.

[6] D. Schuff, Y. Choe, and V. Pai, "Conservative vs. optimistic parallelization of stateful network intrusion detection," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, 2008, pp. 32--43.

[7] B. Haagdorens, T. Vermeiren, and M. Goossens, "Improving the performance of signature-based network intrusion detection sensors by multi-threading," in *International Workshop on Information Security Applications*. Springer, 2004, p. 188.

[8] M. Roesch, "Snort-lightweight intrusion detection for networks," in *Proceedings of the Thirteenth Systems Administration Conference (LISA XIII), November 7-12, 1999, Seattle, WA, USA*. USENIX Association, 1999.

[9] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/ Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[10] P. Wheeler and E. Fulp, "A taxonomy of parallel techniques for intrusion detection," in *Proceedings of the 45th annual southeast regional conference*. ACM Press New York, NY, USA, 2007, pp. 278--282.

[11] A. Le, R. Boutaba, and E. Al-Shaer, "Correlation-based load balancing for network intrusion detection and prevention systems," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks table of contents*. ACM New York, NY, USA, 2008.

[12] V. Paxson, "Bro: a system for detecting network intruders in real-time," in *Proceedings of the 7th conference on USENIX Security Symposium, 1998-Volume 7 table of contents*. USENIX Association Berkeley, CA, USA, 1998, pp. 3--3.

[13] R. Sommer and V. Paxson, "Exploiting independent state for network intrusion detection," in *Proceedings of the 21st Annual Computer Security Applications Conference*. IEEE Computer Society Washington, DC, USA, 2005, pp. 59--71.

[14] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 207--215, 2004.

[15] J. Yu and J. Li, "A parallel nids pattern matching engine and its implementation on network processor," in *Proceedings of the 2005 International Conference on Security and Management. Las Vegas, USA: CSREA Press*, 2005, pp. 375--381.

[16] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333--340, 1975.

[17] M. Roesch, "Snort 3.0 architecture." [Online]. Available: http://securitysauce.blogspot.com/2007/11/snort-30-architecture-series-part-1.html

[18] V. Paxson and R. Sommer, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention," in *2007 IEEE Sarnoff Symposium*, 2007, pp. 1--7.

[19] U. Drepper and I. Molnar, "The native posix thread library for linux," *White Paper, Red Hat, Fevereiro de*, 2003.

[20] V. Jacobson, C. Leres, and S. McCanne, "libpcap: Packet capture library," *Initial public release June*, 1994.

[21] V. Jacobson, C. Leres, S. McCanne *et al.*, "Tcpdump," *available via anonymous ftp to ftp. ee. lbl. gov*, 1989.

[22] G. Insolvibile, "Kernel korner: Inside the linux packet filter," *Linux J.*, vol. 2002, no. 94, p. 7, 2002.

[23] Gianluca Insolvibile, "Kernel korner: Inside the linux packet filter, part ii," *Linux J.*, vol. 2002, no. 94, p. 7, 2002.

[24] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture," in *Proc. Winter'93 USENIX Conference*, 1993.

[25] J. Koziol, *Intrusion detection with SNORT*. Sams Publishing, 2003.

[26] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," 1994.

[27] R. Rehman, *Intrusion detection systems with Snort: advanced IDS techniques using Snort, Apache, MySQL, PHP, and ACID*. Prentice Hall PTR, 2003.

[28] A. Turner and M. Bing, "Tcpreplay," 2005.

[29] K. Vandersloot, "Gnome system monitor," 2001. [Online]. Available: http://freshmeat.net/projects/gnome-system-monitor/

[30] M. Roesch, "Barnyard," 2003. [Online]. Available: http://www.snort.org/dl/barnyard/