PARALLELIZATION OF A SOFTWARE BASED INTRUSION DETECTION SYSTEM - SNORT

A thesis submitted in partial fulfilment of the requirements for the

Degree

of Master of Electrical and Computer Engineer

in the University of Canterbury

by Huan Zhang

University of Canterbury

2011

Abstract

Computer networks are already ubiquitous in people's lives and work and network security is becoming a critical part. A simple firewall, which can only scan the bottom four OSI layers, cannot satisfy all security requirements. An intrusion detection system (IDS) with deep packet inspection, which can filter all seven OSI layers, is becoming necessary for more and more networks. However, the processing throughputs of the IDSs are far behind the current network speed. People have begun to improve the performance of the IDSs by implementing them on different hardware platforms, such as Field-Programmable Gate Array (FPGA) or some special network processors. Nevertheless, all of these options are either less flexible or more expensive to deploy. This research focuses on some possibilities of implementing a parallelized IDS on a general computer environment based on Snort, which is the most popular open-source IDS at the moment.

In this thesis, some possible methods have been analyzed for the parallelization of the pattern-matching engine based on a multicore computer. However, owing to the small granularity of the network packets, the pattern-matching engine of Snort is unsuitable for parallelization. In addition, a pipelined structure of Snort has been implemented and analyzed. The universal packet capture API - LibPCAP has been modified for a new feature, which can capture a packet directly to an external buffer. Then, the performance of the pipelined Snort can have an improvement up to 60% on an Intel i7 multicore computer for jumbo frames. A primary limitation is on the memory bandwidth. With a higher bandwidth, the performance of the parallelization can be further improved.

Keyword: Snort, IDS, Intrusion Detection, Multicore, Parallelization, Pattern Matching.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my supervisors, Professor Dr. Harsha Sirisena from the Electrical and Computer Department in University of Canterbury, Dr. Zhiyi Huang from the Computer Science Department in University of Otago for their guidance, invaluable support and constant encouragement. Their time and effort in providing me with research direction, research ideas and suggestions, real testing computer environment, and other uncountable aspects, is very much appreciated.

My special thanks to Professor Dr. Krzysztof Pawlikowski, Dr. Allan McInnes, Dr. Andreas Willig, and all the members in the network research group from the University of Canterbury for their enlightening advice and wonderful discussion during the weekly meeting, and all members in the system research lab from University of Otago for their help.

I am also grateful for the Electrical and Computer Engineering Department in University of Canterbury and Computer Science Department in the University of Otago for providing me an excellent work environment and equipment during the past years. And special thanks to, David van Leeuwen, Tony Dale and all other computer staffs.

I also want to express my acknowledgements to Industrial Controls South Canterbury to support my studying in different aspects.

Lastly, thanks to any other people and organizations that provide me help and support during my studying and Christchurch earthquake periods.

Glossary

AC algorithm	Aho-Corasick algorithm
ALU	Arithmetic Logic Unit
ANSI	American National Standards Institute
API	Application Programming Interface. It serves as an interface between different software programs.
ARP	Address Resolution Protocol
ASCII	American Standard Code for Information Interchange
ASIC	Application-specific integrated circuit
BDM	Backward Dawg Matching Algorith
BNDM	Backward Nondeterministic Dawg Matching
BogoMips	Bogo Million instructions per second, an unscientific measurement of CPU speed made by the Linux kernel when it boots, to calibrate an internal busy-loop.
BO	Back Orifice, remote administration software
BOM	Backward Oracle Matching algorithm
BM algorithm	Boyer-Moore, a multiple pattern matching algorithm
BP	Bit-Parallelism algorithm
Cilk	A general-purpose programming language designed for multithreaded parallel computing by MIT.
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check, an error-detecting code
DDos	Distributed Denial-of-Service attack
DFA	Deterministic Finite Automaton
DMZ	Demilitarized Zone in Computer Networking
DPI	Deep packet inspection
DoS	Denial-of-Service attack
FDM	Forward Dawg Matching Algorithm
FPGA	Field-programmable Gate Array
GbE	Gigabit Ethernet
HIDS	Host-based Intrusion Detection System
HIPS	Host-based Prevention Detection System
HTTP	Hypertext Transfer Protocol
Horspool	a multiple pattern matching algorithm
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IEEE 802.1Q	Virtual LAN or VLAN
IIS	Internet Information Services, a set of Internet-based services for servers using Microsoft Windows
IP	Internet Protocol
IPS	Intrusion Detection System
KMP	Knuth-Morris-Pratt pattern matching algorithm
LibPCAP	An open source packet capture library
Maotai	A shared memory multiprocessing programming API designed by the University of Otago for multicore computers.
MBNDM	Multiple BNDM algorithm
MBP	Multiple Bit-Parallelism algorithm
MIT	Massachusetts Institute of Technology in US
MPI	Message Passing Interface

MTU	Maximum Transmission Unit
NFA	Nondeterministic Finite Automaton
NIDS	Network-based Intrusion Detection System
NIPS	Network -based Prevention Detection System
OpenMP	Open Multi-Processing, an API that supports multi-platform shared memory multiprocessing programming.
OS	Operating System
OSI	Open Systems Interconnection model
OTN	Option Tree Node
PAC	Parallelized Aho-Corasick algorithm
PCAP	Packet Capture
PDU	Protocol Data Unit in ISO OSI model
PHAC	Parallelized Hashed AC
POSIX	Portable Operating System Interface for Unix. It is a family of related standards specified by the IEEE to define the API.
Qos	Quality of service
RAM	Random-access Memory
RBP	Reversed Bit-Parallelism algorithm
RISC	Reduced Instruction Set Computer
RTN	Rule Tree Node
SBDM	Set Backward Dawg Matching
SBOM	Set Backward Oracle Matching
SDU	Service Data Unit in ISO OSI model
SNMP	Simple Network Management Protocol
Snort	An open source IDS/IPS software, www.snort.org
SPARC	Scalable Processor ARChitecture, a RISC microprocessor instruction set architecture
SQL	Structured Query Language, a database computer declarative language.
TCAM	Ternary Content Addressable Memory
TCP	Transmission Control Protocol
TM	Transactional Memory
UDP	User Datagram Protocol
VODCA	View-Oriented, Distributed, Cluster-based Approach
VOPP	View Oriented Parallel Programming, a shared memory multiprocessing programming API designed by the University of Otago.
VoIP	Voice over IP
WebDAV	Web-based Distributed Authoring and Versioning

I. INTRODUCTION AND BACKGROUND

1 Introduction

1.1 Computer Network System and Its Security

Communication has always been an important part of human development and a spoken language gives people the ability to have face to face communication. Later, people invented writing letters and beacon towers to solve the distance issue and from the 20th century, information gathering, processing and distribution have become the key technology. People invented the telephone and built telephone networks to have long distance and instant communication. The invention of the radio enabled communication to get rid of the cable restrictions and then, the birth and growth of the computer industry accelerated the development of information processing and communication.

Although the development of technology greatly reduces the distance of people, longer distance and more media also bring more security issues for the communicators. Usually, face to face speaking can only be bugged, but the speaker's speech cannot be modified; in letter communication, the message can also be forged by similar handwriting, but it is still difficult. The development of attacking technology came from telephone networks. In order to make unpaid calls, some people tried to find some telephone system bugs, and some experts even broke into the provider's accounting system and modified their records. In the computer network era, hackers also invent computer viruses, worms, spywares, and backdoors to destroy, steal data, block the network, advertise on, and control remote computers. On the Internet, an attacker can do almost everything they want from the other side of the earth. And they can easily hide themselves by controlling one or a group of zombie computers (usually called "Botnet").

On the other hand, most of the intruders started only from personal interest and in the pursuit of higher skills. Even so, now, because of the huge benefits, the attackers are becoming a new industrial chain. They are professional, and sell their products and services on the Internet. A person with very limited skills can buy and download some attack tools to perform attacks, or employ Botnet to storm off the network. Even an unskilled person can easily find some professional attackers to perform attacks for them on the Internet.

A study by Michel Cukier from Clark School, University of Maryland in February 2007 was "one of the first to quantify the near-constant rate of hacker attacks of computers with Internet access—every 39 seconds on average". **[1]**

1.2 Intrusion Detection / Prevention System

Historically, because several major computer viruses have caused huge global losses, people are starting to recognise the importance of anti-virus software. How do they have the virus? Some computers have them by web browning, email and malware. However, some computers, such as servers, without internet access can also get backdoor and data stolen by attacks. A basic firewall can only block some ports on data-link layers and IP addresses on network layers, but cannot see anything on the application layer. Without seeing the detail of the packets, it is difficult to distinguish between a normal access and an attack.

The intrusion detection system (IDS) is a tool which has the ability to read the detail of the network packets. The IDS analyses network packets, captured from the network to identify attacks. The Intrusion Prevention System (IPS) will also automatically block the attacks. Now, IDS/IPS has become a necessary addition to the security infrastructure of most organizations.

Basically, an IDS/IPS can be divided into host-based and network-based. A host-based IDS/IPS (HIDS/HIPS) such as traditional anti-virus software, which operates on information collected from within each individual computer system, can be used to detect or block attacks. However, it has two disadvantages. Firstly, HIDS/HIPS spends too many CPU resources on the host computer, and the benefit can only be received by itself. Secondly, HIDS/HIPS cannot prevent attacks to the network devices such as switches and the intruder can also sniff the packets during data transmission in the network. A network-based IDS/IPS (NIDS/NIPS), which captures and analyses network packets on a network scope can be used to detect all attacks in the network, and all trust domains can receive the benefit.

In general, there are two main intrusion detection techniques: misuse detection and anomaly detection. Misuse detection identifies evidence of malicious behaviour by matching it against predefined descriptions of attacks, or signatures. It has a low false alarm rate and a high miss detection rate. Anomaly detection defines normal behaviour and attempts to identify any unacceptable deviation as possibly the result of an attack. Although anomaly detection has a low miss detection rate, and can detect new attacks, however, it has a high false alarm rate. Misuse detection usually contains a group of signatures, which are used to identify different attacks. The limitation in implementing misuse detection is the high processing load during the traffic detection, which causes a much lower packet forwarding rate in the network than the normal layer-3 switch.

Snort is a lightweight open source network intrusion prevention and detection system (IDS/IPS) developed by Sourcefire **[2]** and is mainly based on a misuse detection mode. It is a well-designed completed IDS and IPS (the inline mode of Snort) solution for a general purpose CPU, such as x86/x64. It supports both IPv4 and IPv6. However, because of its pure software design, it has a relatively slower processing speed compared with ASIC and FPGA.

On the other hand, it is more flexible and has a lower deployment cost, as on the ASIC platform it is almost impossible to make any update and the FPGA is also difficult to upgrade. At this point, many world famous commercial IDS/IPS products are made by a general purpose CPU, such as Cisco.

Currently, Snort is designed as a single process for the detection process, only the reload module uses a separate thread to track the modification of the configuration file and rule update. It was an excellent design last century, but has become a little old now, as the CPU frequency is hard to increase any more, and the performance of a sequential process design is also difficult to improve with the same algorithm. Parallelization is a method which can be used to increase the performance of Snort.

1.3 Multi-core and Parallelization

The earliest computers were programmed by altering the electronic circuitry. In 1945, John von Neumann proposed EDVAC (Electronic Discrete Variable Automatic Computer) architecture which recommended the use of binary as the computer language, and introduced the memory system in the computer to store both instruction sets and data. Today, computers still inherit the John von Neumann architecture. Therefore, the speed of a computer is not only decided by the CPU frequency, but is also based on the speed of the memory and cache.

In 1965, Gordon E. Moore introduced the famous Moore's Law: "the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years". Moore's Law applies to any semiconductor products including a processor's performance and its processing speed, memory capacity, and even the power consumption of the processor which has a similar increase period. History has proved the accuracy of Moore's Law. However, Moore's Law stopped working on the processor's operating frequency from the middle of the 20th century. On 20th October 2004, Intel CEO Dr Craig

Barrett, begged forgiveness for not making a 4GHz Pentium 4, and Intel's product roadmap was substantially revised from the higher frequency development to multi-core chips.

What about the overall increase in performance? Anant Agarwal and Markus Levy indicated a "Moore's Gap" in 2007 and Figure 1 shows the gradually expanding gap from 2002. They analyzed that "the performance scaling fell apart in 2002 due to three factors, namely diminishing returns from single CPU mechanisms such as caching and pipelining, the power envelopes (both active and leakage related), and wire delay." Their suggestions to close the Moore's Gap are multiple full-fledged cores on the same CPU and ample parallelism algorithms. [3]

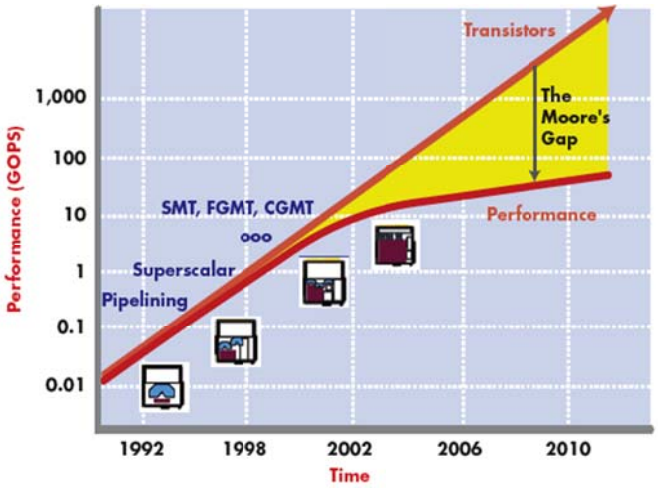


Figure 1: The Moore's Gap. [3]

From the 1960s, people began to develop supercomputers. A supercomputer usually has a much higher computing power than an ordinary computer in the same era, but it does not mean it runs a program faster. A supercomputer is usually built by vector processors, or a cluster of ordinary processors, and these processors are connected not through a high speed bus, but a single processor with a super high operating frequency, instead. And the frequency may be lower than ordinary processors for power efficient reasons. Figure 2 shows the relationship of a processor's power consumption and operating frequency. It shows the processor's power consumption increases exponentially with an increase in the processor's

operating frequency. Therefore, a single sequence program may run more slowly on the super computer.

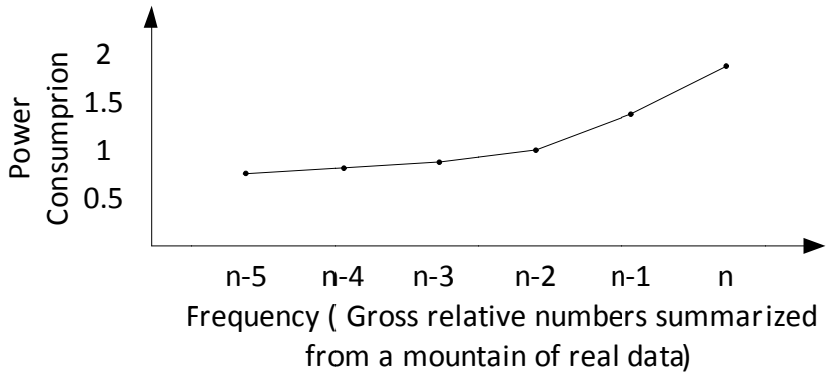


Figure 2: CPU power consumption vs. clock frequency, "In AMD's process, for 200MHz frequency steps, two steps back on frequency cuts power consumption by ~40% from maximum frequency". [4]

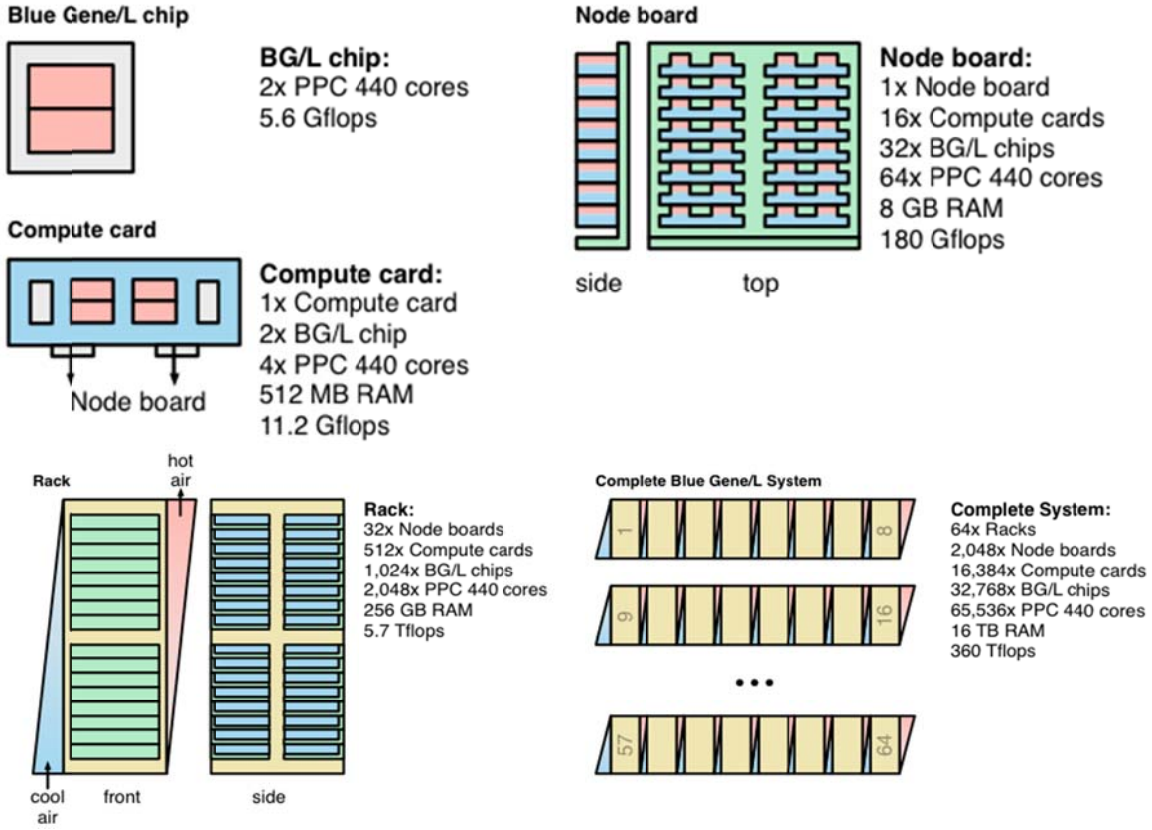


Figure 3: The architecture of a Blue Gene/L. [5]

For example, a Blue Gene/L has 360 TFLOPS computing power. It contains 64 racks and each rack is built by 32 node boards. Each node board includes 16 computer cards and each computer card has two duo-core PPC400 processors. Refer to Figure 3 and Figure 4. There are in total 65536 cores, and the frequency of the processor core is only 700 MHz. Each core

has 5.6 GFLOP. Compared with a supercomputer, an ordinary quad-core processor Intel Core i7 965 XE has 70 GFLOPS in double precision, and each core is about three times faster than a single core in a Blue Gene supercomputer. Basically, supercomputers require the support of the parallelization in software, which can distribute a single huge job to different processors.

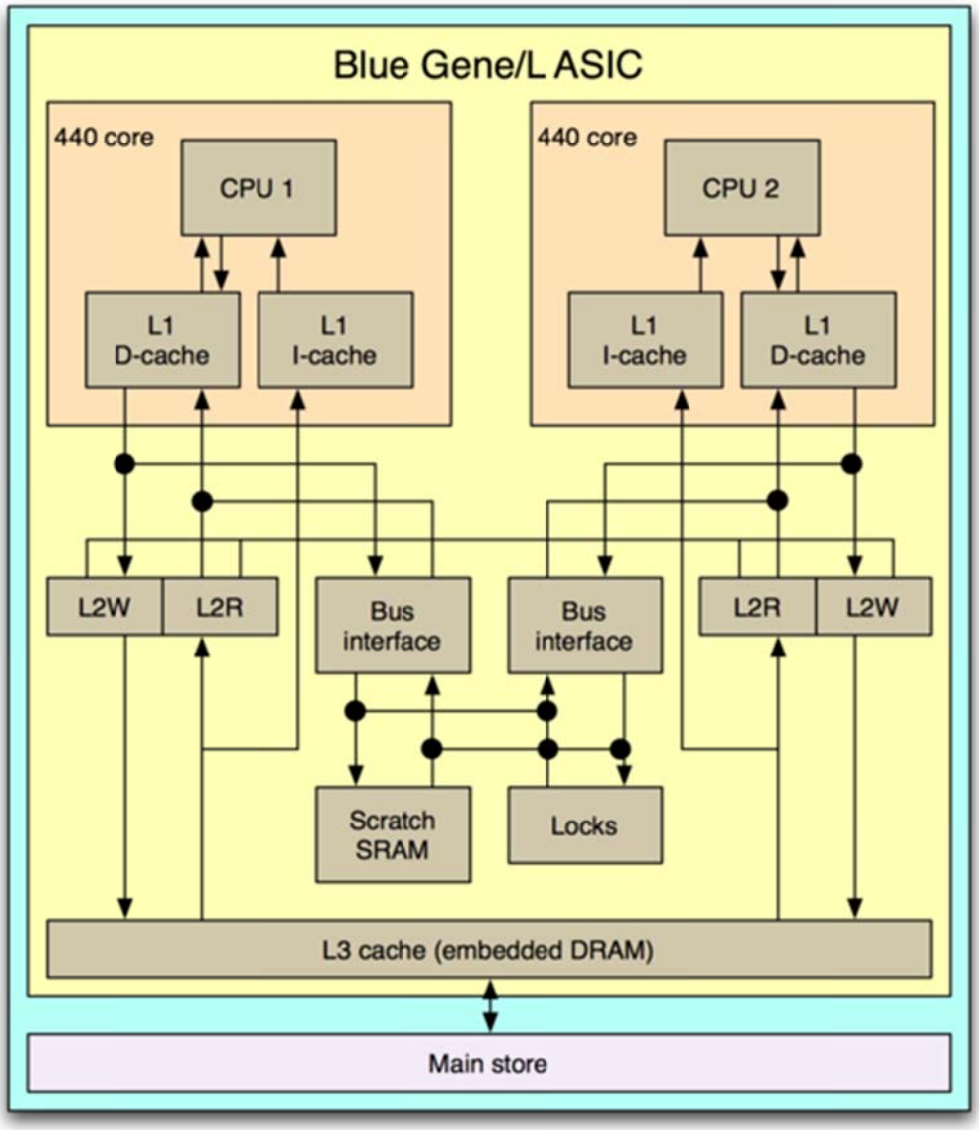


Figure 4: The architecture of a Blue Gene/L ASIC processor. [6]

From the history, we can see that today's super computer will be tomorrow's ordinary computer. As the processors are made smaller, it will be possible to have thousands of cores built in one single processor, and hundreds of processors in one computer. In recent years,

people have been working on integrating multiple cores into one processor to achieve a higher overall performance.

In 2002, Intel introduced its Hyper-Threading technology in Foster MP-based Xeon processors. Actually, it is not a type of multi-core processor as Hyper-Threading duplicates certain sections of the processor, in order to have two "logical" processors to the operating system, which allow the OS to schedule two threads simultaneously. However, it only has one execution unit. The scheduled threads have to queue to be executed by the single execution unit. This technology is also called “multi-thread” by some other companies, such as Sun Microsystems.

Multi-cores computers have real multiple execution units and an individual (L1) cache for each execution unit. It looks like a pizza shop in that there are two chefs but only one oven in the hyper-threading system; there are two chefs and also two ovens in the dual-core system. The multi-core processor puts a number of processors onto a single integrated circuit die, or onto multiple dies, but in a single-chip package they usually share one higher level cache between some of the processors. Some examples are shown in Figure 5 and Figure 6.

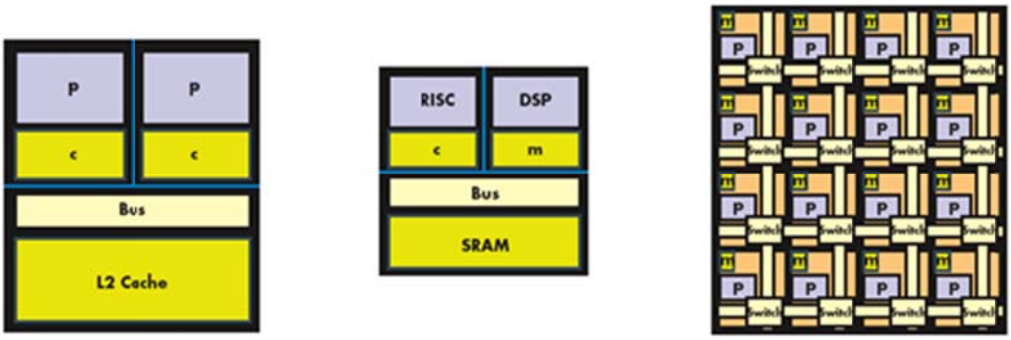


Figure 5: Three examples of multicore implementations, one showing two identical CPUs, one with an RISC CPU and a DSP, and one with 16 identical CPUs. [7]

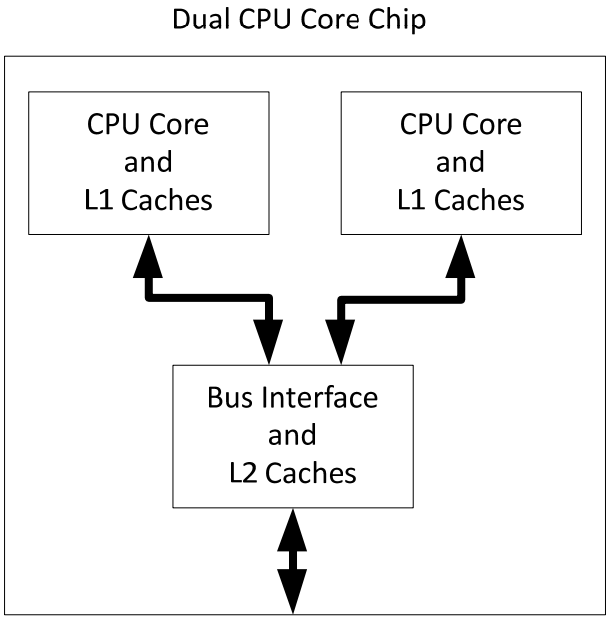


Figure 6: Diagram of a generic dual-core processor, with CPU-local level 1 caches, and a shared, on-die level 2 cache.

Another type of computer system has multiple sockets in one system. The processors in different sockets do not share any cache, but they usually share the memory and memory bus on the same motherboard.

Because a supercomputer is also super expensive owing to the consistency of the design, a computer cluster is designed for powerful computing power. A computer cluster is built by a group of ordinary computers and connected through Ethernet or even through the Internet. A group of retired computers can also achieve very powerful computing power with computer cluster technology. People also expand its use to invent a new computing architecture, and give it a beautiful name “cloud computing”. Many enterprises have super huge groups of computers, called “clouds”. For example, Google had over 450,000 computer clusters in 2006.

In all of the above technology, many people thought the multi-core processor was a good solution. Now, many processor companies, such as Intel, AMD and Nvidia, try to increase the number of processor cores rather than the frequency. In the future, it will be possible to have a

very large number of cores, such as hundreds or even thousands of cores, in one single processor. However, the frequency may not change much. In order to reduce the power consumption, the frequency may be lower than the current frequency.

The running speed of a sequence program is generally decided by the frequency of the CPU. However, a multiple processors computer cannot help to increase the speed of a sequence program. Currently, because the number of processors in a single computer system is not very high, and it normally runs multiple programs simultaneously; different processors can be used to assign different programs by the operation system. With the improvement in technology, the number of processors in a single system has become much higher. Many processors will not get any work from the operation system if all programs are sequence programs. Therefore, parallelization of a sequence program is becoming more important. For many big jobs, it will be easy to parallelize them on their data level. However, for some jobs, it will be difficult to split them up into smaller parts and work on them simultaneously, such as a state machine.

1.4 Research Goals

This thesis will focus on the parallelization of network deep packet inspection programs. As Snort is a world famous open source typical deep packet inspection based intrusion detection system, this thesis will use the parallelization of Snort as an example.

Currently, Snort is still a sequence program. Some types of parallelization have been implemented by researchers on FPGA and some special network processors. However, almost none of them are based on a general processor. This thesis will attempt to find and evaluate some possible methods of Snort parallelization on general computers, such as the x86, x64 and RISC systems. Also, all designs and tests have been carried out in a Linux environment with kernel version 2.6.

At the end of this thesis, the detection performance (throughput) is expected to be increased by parallelization processing. However, as this thesis will not focus on the detection method and signature design, the detection rates and false alarm rates will not change.

1.5 Dissertation Structure

In the following section of this thesis, some background information and some previous work in this area will be discussed. Some mechanisms related to the design in the later chapter will be discussed in detail in Chapter 2. Then, the structure and mechanism of Snort and LibPCAP will be analysed in Chapter 3 and Chapter 4 respectively. Later, the possibility of pattern matching parallelization on a general computer will be discussed in Chapter 5 and a packet level parallelization of Snort and LibPCAP will be evaluated in Chapter 6. Finally, a conclusion of this thesis and some possible future work will be given.

2 Background and Related Work

2.1 Network Packets and “libpcap”library

In general, a network packet is a formatted unit of data transmitted in the computer network. It is defined by the International Organization for Standardization (ISO) in the Open Systems Interconnection model (OSI model) for the network structure. The OSI model divides a communications network system into seven layers, as shown in Figure 7. Each layer is a collection of similar functions that provide services to its upper layer and receive services from its lower layer.

OSI Model			
	Data unit	**Layer**	**Function**
Host layers	Data	7. Application	Network process to application
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data
		5. Session	Interhost communication
	Segments	4. Transport	End-to-end connections and reliability, flow control
Media layers	Packet	3. Network	Path determination and logical addressing
	Frame	2. Data Link	Physical addressing
	Bit	1. Physical	Media, signal and binary transmission

Figure 7: The ISO OSI model. [8]

The IEEE 802 standards also define the data structure in each OSI layer. In an OSI layer, a service data unit (SDU), which is a unit of data, will be passed down to a lower layer, and then be encapsulated into a protocol data unit (PDU) by the lower layer in the sending entity; and the PDU will be decapsulated into the SDU for the upper layer in the receiving entity. In most cases, the SDU corresponds to the PDU of the upper layer. However, they are different in some cases. Sometime, the SDU is bigger, and the protocol in the upper layer requires a smaller PDU, so then this SDU will be divided into multiple PDUs. When a PDU requires a bigger size than the SDU, then multiple SDUs will be combined into one single PDU. The

protocol stack is shown in Figure 8, and a one to one PDU/SDU data structure is shown in Figure 9.

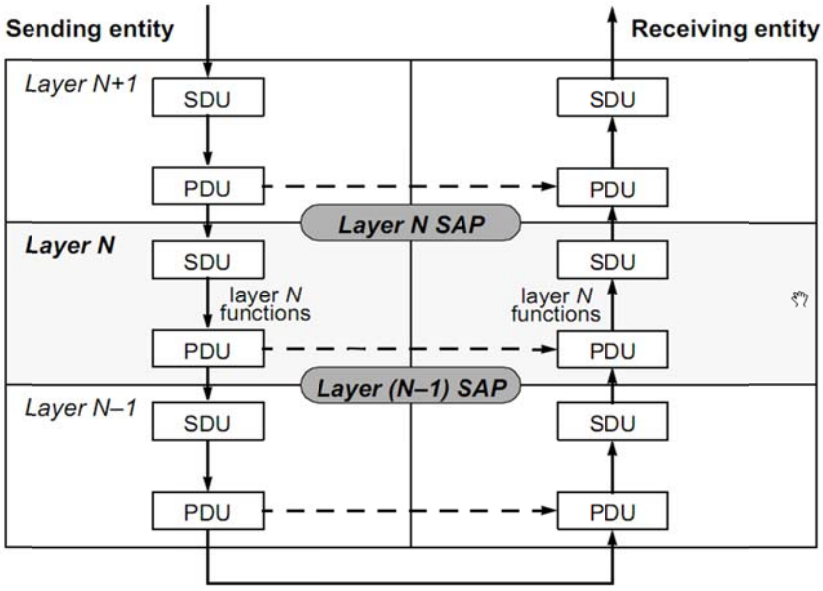


Figure 8: PDU and SDU in a protocol stack. [9]

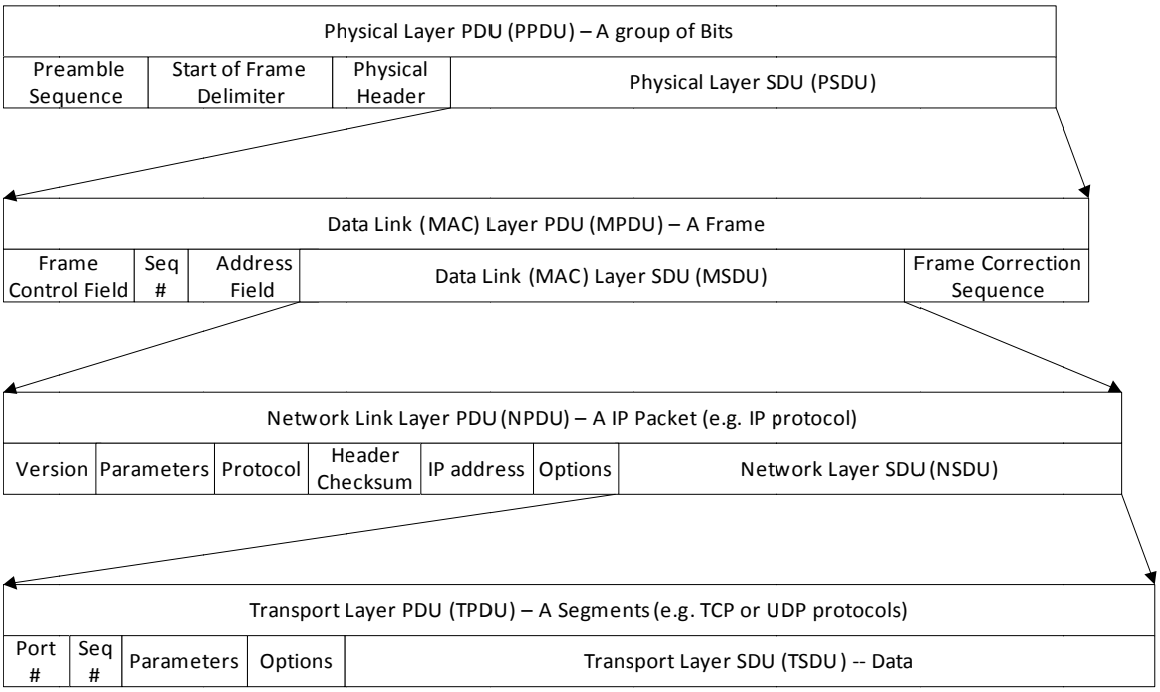


Figure 9: The data structure of PDU and SDU in the first four OSI layers. [10]

In the OSI model, layer 1 generally performs the conversion of logical digital bits and a group of signals, such as electrical signals or optical signals, and transfers it in different coding

theories. A network device usually works from layer 2 to layer 4, e.g. an unmanaged switch is working in layer 2; a router is working in layer 3; and a basic firewall is normally working in layer 4. The deep packet will work in layer 7. However, as layer 5 and layer 6 are not used in the IP protocols, the transport layer SDU is usually the application layer PDU.

Snort uses the "libpcap" library for the packet capture. The data read from the physical NIC card is the PDU of the data link layer. "Libpcap" calls the driver of NIC and gets the SDU of the data link layer. Finally, the output data of "libpcap" is the PDU of the network layer. Snort reads the packets (PDU of the network layer) one by one, and then uses its decode module to get all header parameters of the network layer, transport layer and application layer for the single packet as the header information, and the SDU of the application layer as the packet content. For the packets in which one SDU is not corresponding to a single PDU, Snort will make the packet and process it in the packet reassembly part.

As the IP protocols are the primary layer-3 protocols in the world now, only IP protocols and its family members will be considered in Snort in this thesis.

2.2 Parallelization Algorithms

In general, there are two major techniques to distribute a single job into different processors: multi-processing and multi-threading. Multi-processing technology uses "fork" to create an additional separate process manually in order to execute all later code after the process has been created, like running a new program. The idea of multi-threading technology is dividing a single big job into multiple small tasks, which can be scheduled to be executed by the operating system. Although, on a single core single processor system, the tasks will still be executed by a single processor by a time-division multiplexing, multi-threading can also increase the utilization of a single processor core by leveraging thread-level and instruction-

level parallelism. On a multi-threading and multi-processor enabled system, the multiple tasks will be distributed into different processor cores.

A main difference between multi-processing and multi-threading techniques is that a program running by multi-processing is actually executed by multiple processes in the operations system; but multi-threading will only use one single process to execute the program even on multiple processors. Therefore, different processors will have their own memory and cache space for the same variable name on a multi-processing system; on a multi-threading system, several threads work on the same set of data and also share their cache by default, which has a better cache usage or synchronization on its values.

Sometimes, different processes need to exchange some information between different processor cores. There are generally two methods for this: “Message Passing” and “Shared Memory”.

“Message Passing” is a kind of communication mechanism by which processes can send and receive messages from other processes. It is just like multiple people working on one job together by communicating the progress, ideas and everything related to the job. The type of message includes the form of signals, data packets and so on. Message Passing Interface (MPI) is the most famous “message passing” API for parallel computing. It works on almost any type of platform, including multi-core, multi-thread systems, supercomputers and even computer clusters.

“Shared memory” sets a piece of memory as a public area, which can be simultaneously accessed by multiple processes. It works on a very low level, as the memory operation is the responsibility of an operation system. Linux kernel natively supports some methods to set up a shared memory area, such as “IPC Shared Memory (ipc.h)” and “POSIX Shared Memory

(shm.h)" and "mmap". A shared memory system is relatively easy to use since all processes share a single view similar to memory access to the same location. However, there are some limitations to the "Shared Memory". Firstly, the current computer architecture is not cache coherent. Secondary, the data accessing by different processes will also cause a memory coherence issue. Then, the execution method of the processor will cause "Data Race" and "Deadlock".

Cache memory is a random access memory (RAM) usually on the CPU that a processor can access more quickly than a regular RAM. In the data processing, the processor looks first in the cache (from a previous reading of data), and reads the RAM if the required data is not found in the cache by spending more time. Table 1 shows a general data accessing speed in a normal computer system. Data updated by one processor may be used by other processors, and the change needs to be reflected in the other processors at the same time, otherwise the different processors will be working with incoherent data. In a system in which the cache is not shared, there is usually a delay on the cache coherence.

Table 1: An approximate speed list with data transmission on an Intel Duo-core 2.33 GHz CPU with a DDR2 667 asymmetric dual channel memory system

	Speed (ns)
L1 cache reference	1
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	50
Main memory reference	50
Read 1 MB sequentially from memory	250,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	30,000,000

In addition, an Arithmetic Logic Unit (ALU) of the processor will also not directly use data in a cache; it needs to read the data into some register in the processor and executes them from registers. The register is a small set of data holding places that are part of a computer

processor. It is the closest storage unit from the ALU in the processor. The reading and writing process by different processes from and into the same memory location will cause "Data Race". An example is shown in Figure 10.

	Thread #1	Thread #2	Note
C Statement	X = X+1	X = X+1	Two process try to update the same "X"
Instructions in the right order	LOAD X,R1		X = 0, R1 = 0
	INC R1		R1 = 1
	STORE R1, X		R1 = 1, X = 1
		LOAD X,R2	X = 1, R2 = 1
		INC R2	R2 = 2
		STORE R2, X	R2 = 2, **X = 2**
Instructions when "Data Race"	LOAD X,R1		X = 0, R1 = 0
		LOAD X,R2	X = 0, R2 = 0
		INC R2	R2 = 1
		STORE R2, X	R2 = 1, X = 1
	INC R1		R1 = 1
	STORE R1, X		R1 = 1, **X = 1**

Figure 10: An example of Data Race.

A solution to solve "Data Race" is forcing the atomicity of a group of instructions. Currently, mutual exclusion and transactional memory (TM) [12][13][14][15] can be used to guarantee the atomicity. Mutual exclusion is a traditional pessimistic approach method. The writer has the onus to make sure the shared variables are not being used by some others. A process or thread can lock shared variables before writing or reading, and unlock them at the end of accessing. And all other processes or threads cannot access these locked variables. They usually have to wait or sleep until the designed variable is available again. In the case of more than two processes or threads working on one shared variable, the possibility of accessing crash will increase exponentially with the increase in the number of processes or threads. "Semaphore" and "Mutex" are two API of mutual exclusion. However, when locks are nested, deadlock can happen. Deadlock is a situation where threads or processes are waiting for locks that will never be released.[18] An example is shown in Figure 11.

Note (Thread #1)	Thread #1	Thread #2	Note (Thread #2)
	Acquire lock 1	Acquire lock 2	
Waiting for lock 2 forever as lock 2 has already been locked by Thread 2, but Thread 2 is also waiting.	Acquire lock 2	Acquire lock 1	Waiting for lock 1 forever as lock 1 has already been locked by Thread 1, but Thread 1 is also waiting.

Figure 11: An example of Deadlock.

Transactional memory (TM) is an optimistic approach which was designed in the late 1980s, and is reaching maturity now. Differently from "locking" techniques, TM allows multiple threads or processes to perform modifications to the same shared memory without regard for any other threads or processes, and it is "Data Race" and "Deadlock" free. However, it records every read and write by different threads and processes in a log. The idea is a little similar to a journal file system, such as NTFS, ext3, JFS and ReiserFS. The reader will have the onus of verifying if any others have concurrently "write" access to the same memory when it completes an entire transaction. If there is no other "write" access, then the transaction will be committed as permanent, otherwise, it will roll back, and revise the transaction. As no threads or processes are required to wait for access to any resource, multiple threads or processes can continue to work concurrently. In the case of no modification in the shared memory, the performance of TM is very high. However, there are usually quite a number of repeated transactions in the real practice. In addition, the management overhead, e.g. maintaining the log and committing transactions, is still quite high in software based TM.

In a multi-thread/ multi-process environment, it will be hard to control the progress of each thread or process. For example, when two threads are assigned a job to acquire a lock on the same shared variable at the same time, both of them will have a chance to acquire the lock first, but the shared variable can only be locked by one thread. The other has to wait. As the order of which thread locks the shared variable is random, it is called a "determinacy race".

Some synchronization methods can be used to solve this issue, such as “barrier” (or “fence” sometimes) and “messages”.

As the basic pessimistic approach shared memory has a few limitations and difficulties in use, some APIs are designed to avoid the “Data Race” and “Deadlock”, and simplify the synchronization process. OpenMP is an API which makes the implementation of shared memory parallel programming much easier. However, it is only designed to optimize the performance of “for” loop in the program. [11][16]

The University of Otago has designed a series of View Oriented Parallel Programming (VOPP) [17]-[22] APIs, which include VODCA [23] and Maotai [24][25]. VOPP is a process based shared memory parallel programming API, and it can automatically prevent “Data Race” and “Deadlock”. VODCA is specialized for a cluster system; and Maotai is specialized for a multi-core computer.

Cilk was designed by MIT in the middle of the 1990s and is a multithreaded parallel programming language based on ANSI C. It is designed for general-purpose parallel programming, but is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style. [26]

Different API and penalization methods have different performances. However, Gene Amdahl said that “the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program”. Amdahl's law can be used to predict the theoretical maximum speedup using multiple processors in parallel computing. Generally, the speedup of a program is limited by the time needed for the sequential fraction of the program. The formula is shown in Equation 1, and a speedup graph vs. the number of processors is shown in Figure 12.

$$Speedup = \frac{1}{1-f} = \frac{1}{r_s + \frac{r_p}{n}}$$

Where $r_s + r_p = 1$, r_s represents the ration of the sequential portion in one program, and

$$f = \left(1 - \frac{1}{n}\right) \cdot r_p.$$

Equation 1

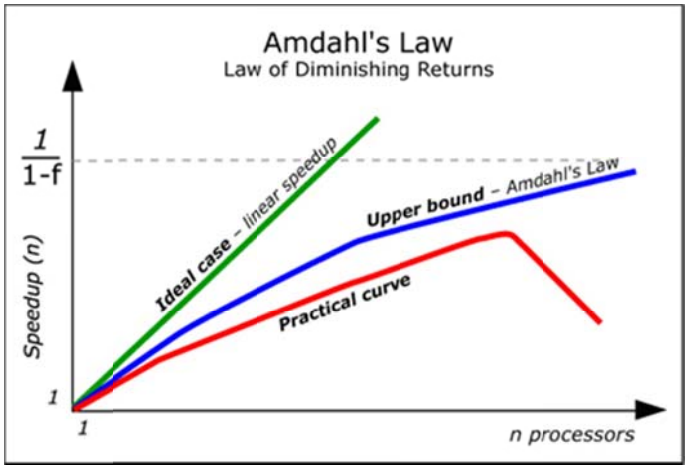


Figure 12: The data structure of the PDU and SDU in the first four OSI layers. [27]

2.3 Pattern Matching Algorithms and Detection Engine

Deep packet inspection is actually a type of pattern matching process and pattern matching is also the primary part of the detection engine of IDS. A highly efficient pattern matching algorithm will lead to a high performance IDS.

In general, the string matching includes exact string matching and approximate string matching (string matching allowing errors).

The one-dimensional exact string matching consists of [29] (1) Single String Matching, the most basic string matching algorithm, searches for a single pattern (p) with "m" characters in a text string (t) with the length "n". Both strings are built over a finite set of characters (σ) denoted as alphabet (∑). (2) Multiple String Matching searches for a set of pattern P

simultaneously. (3) In Regular Expression Matching, the pattern is described by the regular expression. This is the most complex matching, a single regular expression can describe thousands of different normal patterns. Regular expression is now widely used in the IDS.

2.3.1. Single Pattern Matching Algorithms

The single string matching algorithm is the kernel of the general string matching and includes regular expression matching and approximate string matching. Figure 13 shows some historical main string matching algorithms. In general, the hundreds of single string matching algorithms in the world can be classified into five categories: (1) Brute Force; (2) Hash; (3) Automata; (4) Sliding Window (Prefix & Suffix); (5) Bit-Parallelism.[29][31]

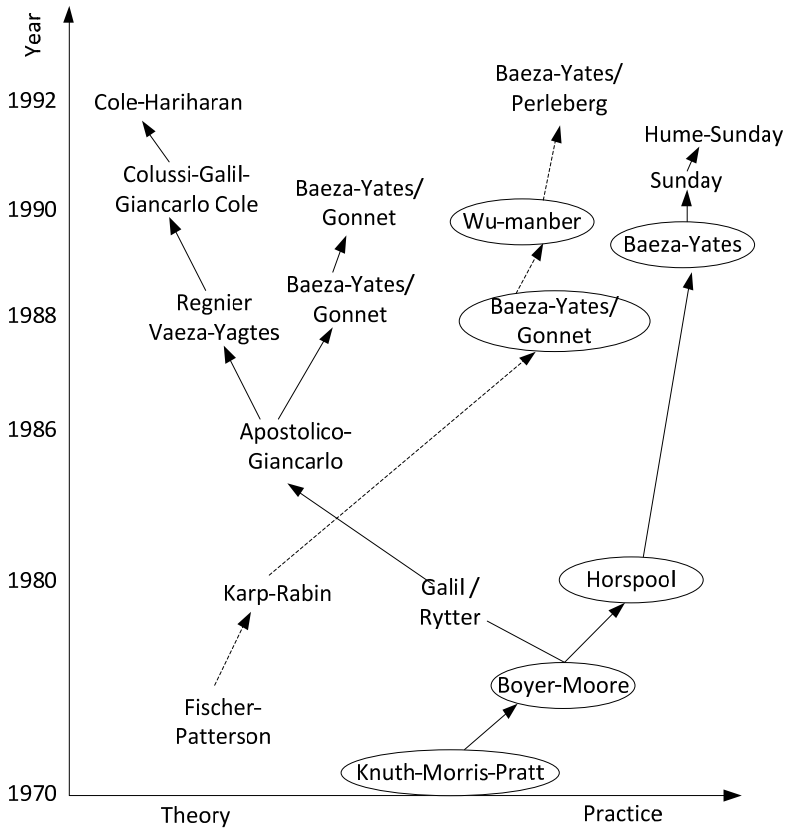


Figure 13: The development history of pattern matching algorithms. [30]

2.3.2. Brute Force Algorithm

The Brute Force algorithm [31] is the most basic search algorithm. It checks the pattern at all positions in the text between 0 and n-m, and shifts the pattern by exactly one position to the right after each attempt. It has a fixed time complexity O(m x n).

2.3.3. Hash Algorithm

Hashing provides a simple method to avoid a large number of character comparisons. Usually, a small change on the string will make a big difference to the hashing value. The Karp-Rabin algorithm [31][32], is one of the most famous search algorithms based on hashing. It calculates the hashing value of a pattern with length m and the text for every continued m characters. If two hashing values are same, it will compare every character of these two strings to confirm the match. It has a time complexity O(m x n).

2.3.4. Automata Algorithm

An automaton [30][31][33] is a mathematical model for a Finite State Machine. An automaton can be represented by $A = (Q, \Sigma, I, F, D)$, which represents a finite state "Q" with an initial state "I" and a final state "F". Transitions between states can jump on the input of alphabet "Σ" and empty input "ε". Then the transition function "$\mathcal{D}$" can be defined by each state $q_k \in Q$ for each input $\alpha \in (\Sigma \cup \varepsilon)$.

Generally, there are two types of automata: the Nondeterministic Finite Automaton (NFA) and the Deterministic Finite Automaton (DFA). If a state q can jump to multiple different states with only one input α, or empty input ε exists in the transition, then this automaton is NFA. The transition function $\mathcal{D}$ of NFA is denoted by the set of triples Δ. Otherwise, the automaton is DFA, and the transition function $\mathcal{D}$ denoted by a partial function $\delta: Q \times \Sigma \rightarrow Q$. In the area of pattern matching, it is easier to obtain the NFA of a string. However, maybe there are multiple active states in an NFA and only one active state in a DFA. Therefore, DFA can usually have a faster search speed for pattern matching and all NFA can be converted into DFA. [29][33]

The Forward/Backward Dawg Matching Algorithm (FDM/BDM) [31] is a famous typical DFA based algorithm. It tries to compute the longest factor of the pattern ending at each position in the text. However, its complex process slows down its overall speed.

2.3.5. Sliding Window Algorithm

The Sliding Window algorithm [29] uses a virtual window with a length m on the test to perform matching. There are two general ways to attempt the matching: (1) based on the prefix; (2) based on the suffix.

Knuth-Morris-Pratt (KMP) [31][34] and Boyer-Moore (BM) [29][31][35] are the most famous search algorithms. Many other algorithms have evolved from them. The primary difference is that KMP is a prefix based approach, and BM is a suffix based approach.

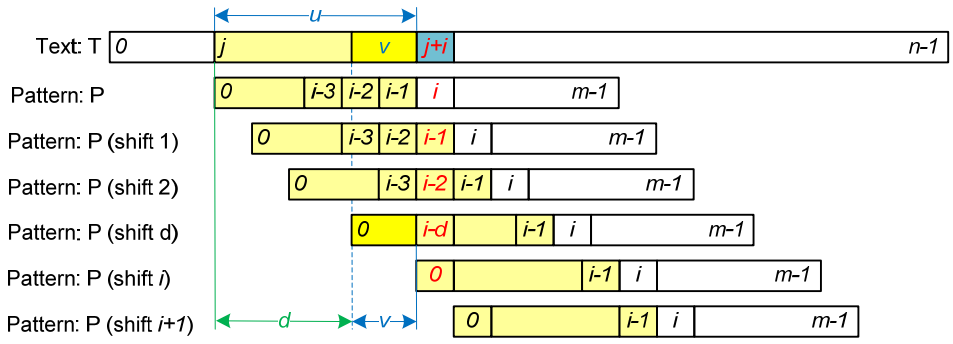


Figure 14: Shift in the Prefix Algorithm (Knuth-Morris-Pratt).

The Prefix algorithm follows a tight analysis of the Brute Force algorithm. It tries to find the length of the longest prefix of the pattern that is also a suffix of the text. Figure 14 shows it searching for a pattern p in the text T. When a window is positioned on the text T[j ... j+m-1], assume that the first mismatch occurs between T[j+1] and p[i]. Let T[j ... j+i-1] = p[0 ... i-1] = u, then the next characters T[j+i] ≠ p[i]. When shifting, the Brute Force algorithm will only shift one character, but we can expect if d characters have been shifted, a prefix v = p[0 ... i-d-1] of the pattern may match some suffix of the portion u. For example, if only one character

has been shifted, T[j+i] ≠ p[i-1] or T[j+i] = p[i-1] but T[j+i-1] ≠ p[i-2], it is impossible for the pattern p to match the text T[j+1 ... j+i]. Therefore, if we can know it at the beginning, it is not necessary to waste time shifting only one character. Only if p[0 ... i-d-1] = T[j+d ... j+i-1], is it possible to have p[0 ... m-1] = T[j+d ... j+d+m-1]. If no such border exists the length of the pattern characters can be shifted directly. Different prefix algorithms use different ways to find the best d. The best algorithm can use the easiest algorithm to find the biggest d. For example, KMP uses p[i] to calculate d. The pre-process session has time complexity O(m), and the worst case of the search phase has time complexity O(m+n). In addition, different algorithms may also have a different matching order of T[j ... j+i-1] and p[0 ... i-1]. For example, a general algorithm, such as KMP, will match in the order from left to right. The Apostolico-Crochemore algorithm matches the second character (T[j+1] with p[1]) first, and the first character (T[j] with p[0]) last. [29][31][34]

The suffix search algorithm scans the characters of the pattern from right to left and begins with the rightmost character. Similarly to the prefix algorithm, it ignores some unnecessary matches, and tries to find the biggest shifts. BM is a typical suffix based algorithm. Because it is difficult to calculate the biggest safety shift directly, BM defines three ways to shift the pattern (Figure 15), and any of them can shift the pattern without missing any occurrence. Firstly, if T[j+i+1 ...j+m-1] = p[i+1...m-1] = u and T[j+1] ≠ p[i], it tries to find a shift d1, which can have another u in p[0...m-d1-1] from p[0]. Secondly, when the suffix u does not appear in any other position of pattern p, but a suffix v of u exists which is the same as the prefix of the pattern, BM will look for a shift d2, which has the longest prefix v of pattern p that is also a suffix of u. Thirdly, with a shift d3, if p[i-d3] = T[j+i] and if there is not any character equal to T[j+i] in the range p[i-d3+1...m-1], we can directly perform this shift safely. If there is not T[j+1] in the pattern, we can directly shift the pattern by i+1. All three

shifts are calculated in the pre- pre-process session and the results are stored in different tables, and then the longest shift will be used in the search process. [29][31][35]

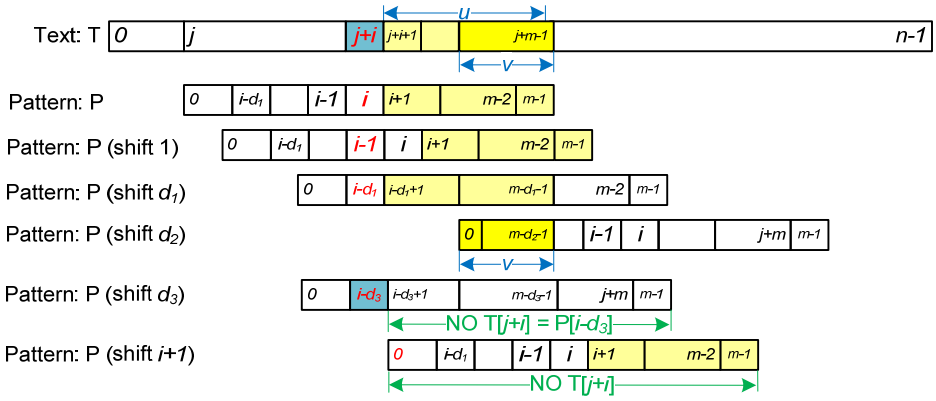


Figure 15: Shift in the Suffix Algorithm (Boyer-Moore).

Although the first two methods may have a longer shift sometimes, they require much more time on the calculation in the pre- pre-process session. Some algorithms are designed to simplify or modify the BM algorithm, such as the Turbo-BM, the Apostolico-Giancarlo algorithm, the BM-Horspool, the Quick Search algorithm, the Tuned BM, Zhu-Takaoka algorithm, the Berry-Ravindran algorithm, the Smith algorithm and the Raita algorithm. BM-Horspool (Horspool) [29][31][36] is the most famous one. It only reserves the third shift method of BM owing to its having a good balancing on the shift length and its complexity of calculation. Therefore, it is faster in most cases. Until now, Horspool is still acknowledged as the fastest single pattern matching algorithm in general cases in the world. Currently, Snort uses BM as the default matching algorithm, and supports Horspool through plug-in.

In addition, the Backward Oracle Matching algorithm (BOM) [29][31][37] imports the automaton technique into the suffix window shift algorithm. It creates a simple DFA with m+1 states and m to 2m-1 transition in the pre- pre-process session. Figure 16 shows an example of the DFA build by BOM. The search engine is similar to BM. The search will be carried out backwards in each window following the DFA. When a mismatch (a non-existent

state in the DFA) has been detected, the window will be shifted to the next character of the mismatch, which is shown in Figure 17. Because BOM has a simpler DFA, the search speed is faster than FDM/BDM. It has a competitive speed for a large pattern length. However, the speed is still slower than the Horspool and Bit Parallel algorithms for a small size of pattern.

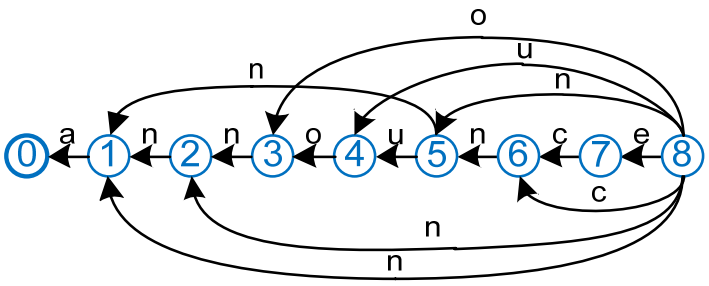


Figure 16: A DFA built by BOM for Pattern "announce". [10]

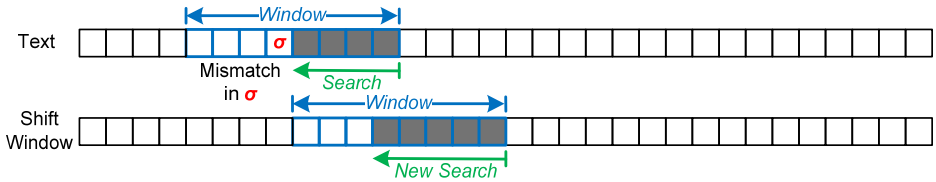


Figure 17: Shift of the search window after the failure of the search by BOM. [11]

2.3.6. Bit-Parallelism Algorithm

All the above matching algorithms have evolved from the Brute Force algorithm, which stores two characters (one from the pattern, and one from the test) into two variables, e.g. unsigned char (8 bits), unsigned int (16 bits) and unsigned long (32 bits) in ANSI C. Actually, the variables are stored as binary in the memory. For example, an English alphabet which is encoding in ASCII only uses 7 bits. An "unsigned char" is defined as 8 bits in the memory. Each unsigned char can be used to store an English character. Then the computer will compare the two variables. If the processor is 32-bit, which can process data and memory addresses with the width of registers at 32 bits, each unsigned char only uses the last 8 bits in the processor bus, the remaining 24 bits of the bus are not going to be used. Then the two 8

bits unsigned chars can be processed simultaneously, and compared all in a single operation. (Figure 18) However, it wastes 75% of the processor bus.

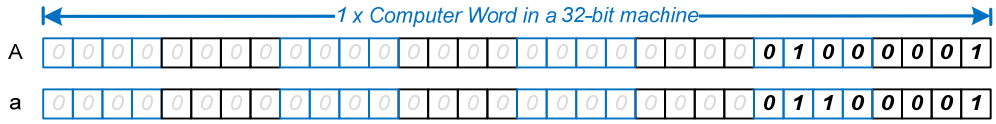
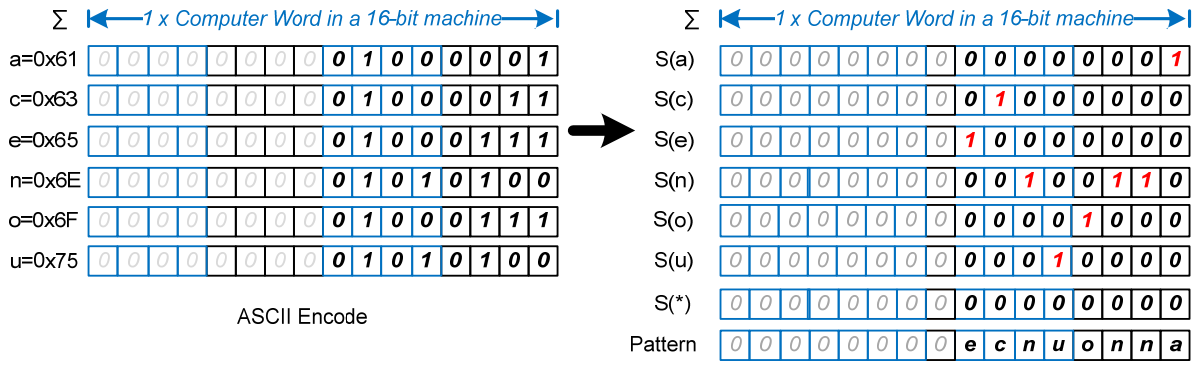


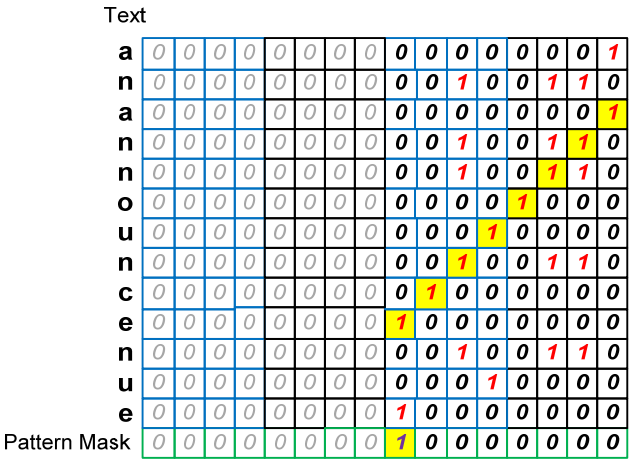
Figure 18: Comparison of two ASCII characters in the memory.

The Bit-Parallelism algorithm **[29-[31][38]** is different. Its purpose is not to look for the biggest window shift, but it tries to use the simplest and the fastest way to process all comparisons by using a bitwise technique. It uses a bit to describe if a character with any encoding format is in the position of a pattern. Therefore, it requires a size of alphabet $\sigma \times m$ (pattern length) table to describe a single pattern.

For example, the processing of searching pattern “announce” in the text “anannouncenue” in a 16-bit computer is shown in Figure 19. Firstly, the pattern will be decoded into an “S” table in the pre-process session; all the matched characters are marked by 1. By calling over each character in the text from left to right using the value in S table, a continual match (“1” with the yellow background) from the 3rd to 12th character of text can be found in Table R. Actually, it is not necessary to keep the whole Table R. By keeping one variable, the next result can be calculated by Ri+1=((Ri<<1) | 1) & Si+1 where i is the current position of the text and R0=1(word length). Then check if the result of the m (the length of pattern) bit is 1. Because the example uses one shift and one AND operation to do the matching, it is also called the Shift-AND algorithm. Similarly, by identifying the matched characters by 0 and unmatched characters by 1, the equation can be changed to Ri+1=(Ri<<1) | Si+1 where R0=0, which is called the Shift-OR Algorithm. Shift-OR may be slightly faster in some processors owing to the computation of Ri+1 having been reduced to two operations, shift and OR (without OR and 1 at the end).



Call over each character in the text from left to right using the value in S table:



Shift above table, compensate with 1 at the end, without AND:

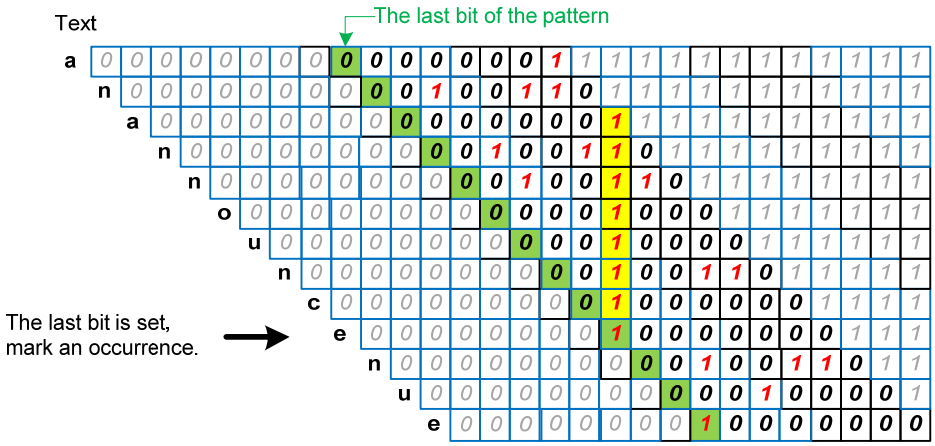


Figure 19: Searching pattern "announce" in the text "anannouncenue" in a 16-bit computer by the Bit-Parallelism (Shift-AND) Algorithm. Characters are encoded in ASCII

However, there is a limitation on the Bit-Parallelism algorithm owing to the length of the computer word. The total pattern m cannot be longer than the computer word. For example,

the maximum pattern length that can be processed is 32 characters by a 32-bit computer, and 64 characters by a 64-bit computer. Although a longer variable can be simulated by multiple computer words, it cannot be processed simultaneously. Therefore, the simulation will slow down the whole matching process. Hence, the Bit-Parallelism algorithm is good at a short pattern with a small alphabet. With a large alphabet, the matching possibility will be low. A Sliding Window Algorithm, such as Horspool, which requires less comparing, and has a bigger window shift, will be faster.

If it is assumed that the pattern length is no longer than the computer word size, the time complexity is O(m+ σ) for the pre-processing, and O(n) for the searching.

In addition, the Bit-Parallelism algorithm can also be processed backwards. Backward Nondeterministic Dawg Matching (BNDM) [29] uses a similar search approach with BDM, but the factor is searched by bit-parallelism. Its pre- pre-process session is the same as with the normal Bit-Parallelism algorithm (Shift-AND). However, it makes a window on the text during the search. The length of the window is the pattern length m. The first window is from the left of the text and it calls over the characters from right to left in each window. If it loses all matches in a single try (R = 0), the window will shift by m. If it finds the first character of the pattern (R= 1 0(m-1)), the window will shift to the first character of the pattern. Therefore, it can have some jumping during the search. However, the expense is in the more computing operations in the search process (not just two operations: shift and OR). Hence, it is good at a slightly bigger m compared with a normal Bit-Parallelism algorithm.

2.3.7. Multiple String Matching Algorithms

The multiple string matching is an extension of the single string matching algorithm. It is designed to search multiple patterns simultaneously. Most techniques of multiple string matching are based on single string matching algorithms. There are some famous algorithms:

the Aho-Corasick algorithm, the Commentz-Walter algorithm, the Set Horspool algorithm, the Wu-Manber algorithm, the Set Backward Dawg Matching (SBDM) algorithm, the Set Backward Oracle Matching (SBOM) algorithm, the Multiple Bit Parallelism (MBP) algorithm and the multiple BNDM (MBNDM) algorithm. [29]

The Aho-Corasick algorithm **[29][39]** is an extension of the KMP algorithm. It uses an Aho-Corasick (AC) automaton as the pattern which is built by the multiple patterns. An example of an AC automaton is shown in Figure 20.

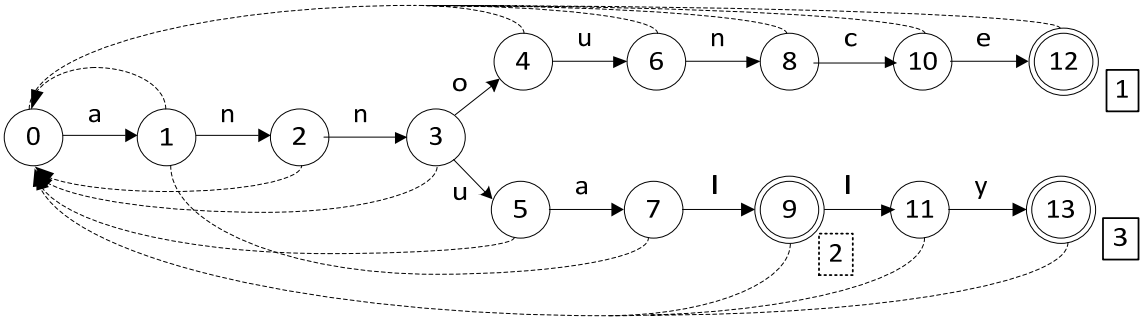


Figure 20: An AC automaton of pattern P = {announce, annual, annually}. Double-circled states are terminal. [11]

The Commentz-Walter algorithm **[29]** is an extension of the BM algorithm, and it is the first sub linear multistring matching algorithm. To avoid skipping any occurrence, the size of a window is set to the minimum length of all sub patterns l_{min}. An example is shown in Figure 21. The shift size of the window will be calculated in the same way as with the BM algorithm.

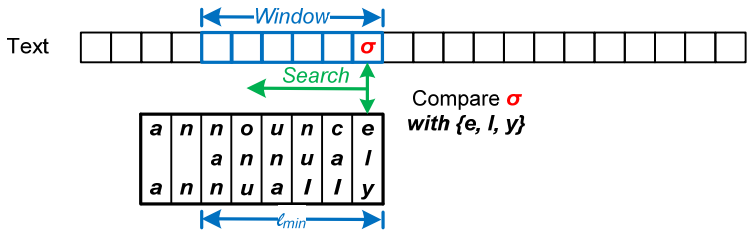


Figure 21: The Sub linear Multi-string Matching algorithm with example patterns P = {announce, annual, annually}.

The Set Horspool algorithm [29] is similar to the BM-Horspool and is a simplification of the Commentz-Walter algorithm. It uses the simplified shift calculation to get the window shift which is exactly the same as the Horspool. In practice, although it has a faster speed than the Commentz-Walter algorithm, its high probability of finding each character of the alphabet in one of the patterns decreases the window shift. Then, the Wu-Manber algorithm is designed to decrease the probability based on the Set Horspool algorithm.

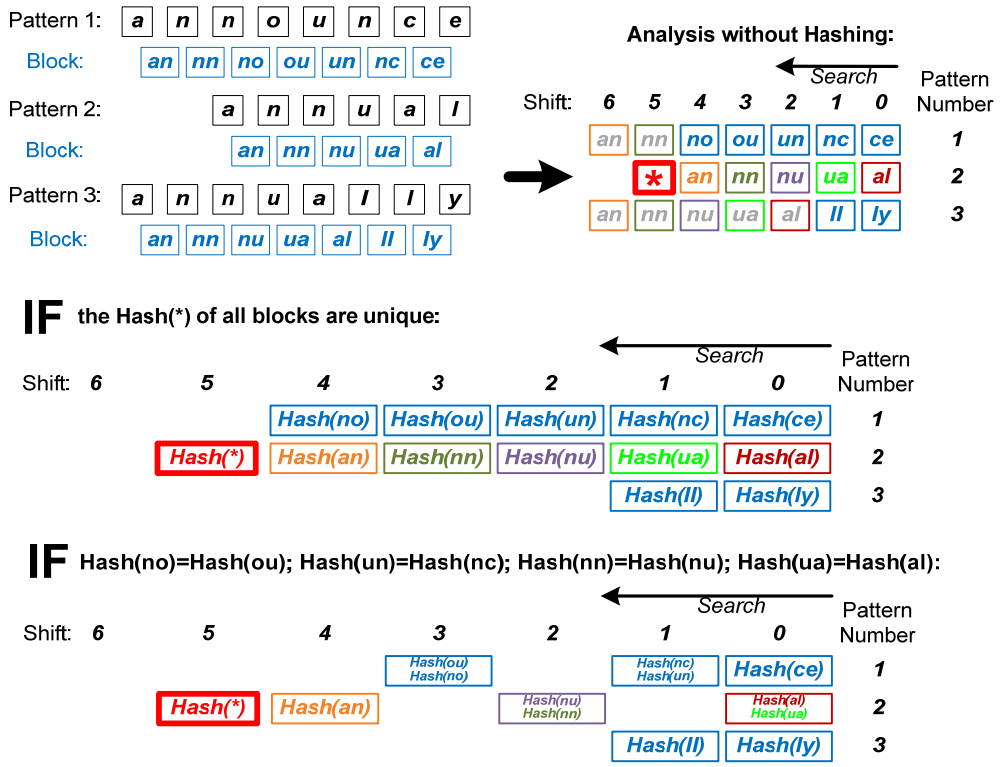


Figure 22: The pattern analysis of the Wu-Manber algorithm with example patterns P = {announce, annual, annually} and the length of the block = 2.

In the usual algorithm, the smallest unit is the character. Wu-Manber [29][40] groups the characters and the comparing units (not the shift unit) become a block of a few characters. Then the matching probability will decrease to σ(-B), where B is the length (the number of characters) of the block. And the difficulty will increase to σB different blocks, which means requiring much more memory when B is big. To solve the memory problem, Wu-Manber applies a hashing function on all possible blocks and different blocks are identified by their

hashing value. The search engine will only compare the hashing value of block and window. The shift step is still calculated in the same way as with Set Horspool, which will try to find the rightmost occurrence. The maximum shift size is l_{min}–B+1. Because the hashing value of different blocks may be equal, when an occurrence is found (shift size is 0), the block of original characters should be checked again. Then a new shift with only one step will apply on the window on the text. An example is shown in Figure 22. Wu-Manber has a fast search speed for a big alphabet (e.g. bigger than 10) with not very many multiple patterns (e.g. 100). [29]

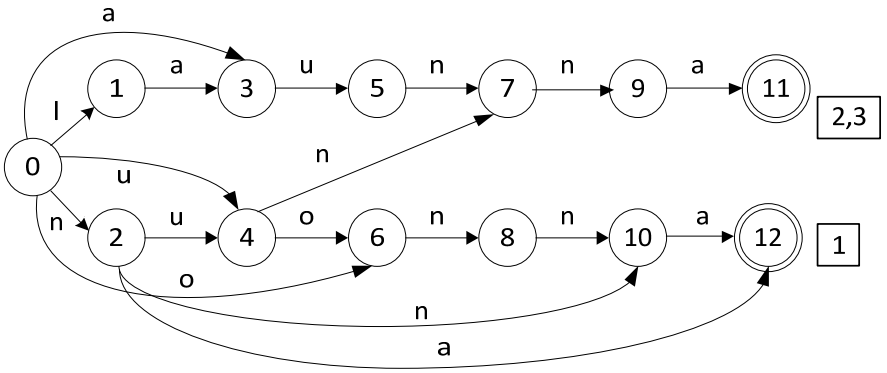


Figure 23: The automaton of SBOM for example patterns P = {announce, annual, annually}. The factor oracle of the reverse set Plmin = {announce, annual }. [11]

The Set Backward Oracle Matching (SBOM) algorithm [29][41] has the same search algorithm as BOM, the only difference is the automaton. The automaton is constructed on a set of strings like the Aho-Corasick, however, it creates an additional transition from each state on the path to the original lead state instead of jumping to the next state. And the length of the automaton does not have the full length of the patterns. It will only keep the prefixes with length l_{min} of all multiple patterns. If there are r multiple patterns, the maximum state of the automaton is $r \times lmin$. An example of the SBOM automaton is shown in Figure 23. During the searching, when an occurrence is found, the search engine will verify the entire

current pattern against the text, and shift the window by 1. SBOM is fast for a super large number of patterns or a small alphabet. [29]

Bit Parallelism can handle parallel characters in a string. The Multiple Bit Parallelism (MBP) algorithm [29] can be seen as adding more characters on the single pattern, and they can be processed at the same time. The change is the pattern mask. In the single pattern, there is only one "1" in the pattern mask to identify the length of the pattern; in the multiple pattern algorithm, there are multiple patterns of "1" to identify the position of the last bit of each pattern. However, it still inherits the pattern length weakness from the BM algorithm, and it has been magnified. The total length of all patterns cannot exceed the size of the computer word; otherwise the search time will have multiplied. A pattern store example is shown in Figure 24.

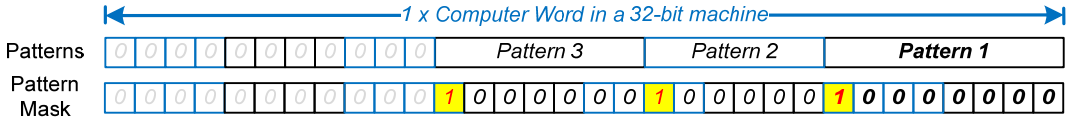


Figure 24: Three patterns in a 32-bit computer for the MBP Algorithm.

The Multiple BNDM (MBNDM) algorithm [29] will resize all patterns to l_{min}, and put all patterns in reverse order. This is similar to the SBOM, if the text matches a prefix. The entire string will be verified against the text. The limited size of the computer word can store few more patterns. A pattern store example is shown in Figure 25.

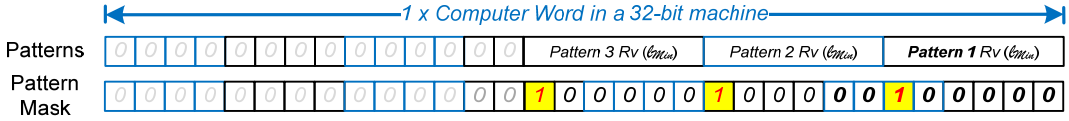


Figure 25: Three patterns in a 32-bit computer for the MBNDM Algorithm.

2.3.8. Snort String Matching Algorithms

Snort officially uses Aho-Corasick (AC) for multiple pattern matching. Furthermore, some new algorithms have been designed based on the traditional string matching algorithms mentioned in the previous sections.

Yaron Weinsberg from the Hebrew University of Jerusalem designed a novel multiple pattern matching algorithm called “Ternary Content Addressable Memory” (TCAM). It requires a special memory chip called TCAM which can store three values: “one”, “zero”, and “don’t care”. Then it uses an algorithm similar to Brute Force to perform the matching. [55]

Piti Piyachon implements some modified Aho-Corasick algorithms on network processors. He gives three different schemes: Bit-level, Byte-level, and Bit-Byte-Level. [57] The Bit-level AC represents each character by 8 bits, and each bit will be an input of the corresponding bit-level state machine. For example, if an input stream is “announce”, the bit 0 sequence of this stream is “10000000”, and its bit 1 sequence is “01100000”. There are in total eight state machines corresponding to 8 bits of each character. A match is declared only when all eight machines reach a common state. The byte-level AC declares multiple characters as one single unit to build the state machine. Bit-Byte-level AC adds the previous two options together, which use all bits of a string as one single unit.

2.4 Current IDS (Snort) parallelization

At this point, Snort is officially a sequential program. A fake parallelisation has been carried out based on the network interface. With multiple network interfaces, multiple instances can be created for each individual interface with a low overall performance.

Recently, much research has been carried out to parallelize the IDS, and most of it is based on Snort owing to its GPL licence, and the well-designed modular structure.

As the pattern matching engine is the kernel of an IDS system, over 60% of the CPU work load on average is spent on it.[9] Some people are trying to improve the performance of the pattern matching engine, however, most of them are trying to move this part into a hardware platform. For example, Jakub Botwicz has implemented the Karp-Rabin algorithm in hardware architecture, which increases the overall throughput of Snort to about 2Gbps. [43] Sarang Dharmapurikar and John W. Lockwood have designed a hardware-implementable pattern matching algorithm for a content filtering application. The algorithm is based on a memory efficient multi-hashing data structure, like a hardware implementation of Wu-Manber. [45]

With a similar running frequency, a specified hardware implementation is usually executing faster than a general-purpose processor. Therefore, a large number of researchers choose to use a hardware implementation for IDS, especially the FPGA. There are quite a few different approaches. Apart from the hardware implementation of a pattern matching engine [43][45][58], a popular method is signature header classification, for example, “Ternary Content Addressable (TCAM)”. [46][47][48] Another approach is implementing reconfigurable pattern matchers and filters. [49] - [53] In addition, memory optimization and hashing are other popular paths to achieve a low cost system. [54]

However, the hardware implementation still has many disadvantages. The main disadvantage is the difficulty of re-programming and some people consider a compromise solution, which uses a hybrid system instead of the pure software or hardware system. Some frequently updated parts are put in the software system and the rest in the hardware system. For example, Young H. Cho and William H. Mangione-Smith used a hybrid Snort in 2005. [67]

There is only a small amount of research for IDS on a general multi-core platform. Zhuojun Zhuang and his group from Shanghai Jiao Tong University introduced an abstract NIDS

model and a related resource scheduling approach, which is shown in Figure 26. It added a packet scheduler into the original snort module, and the scheduler model is protocol-oriented, which classifies packets into a different processing sensor by protocol type. A traffic auditor tracks the speeds of all flows by scheduler. [56]

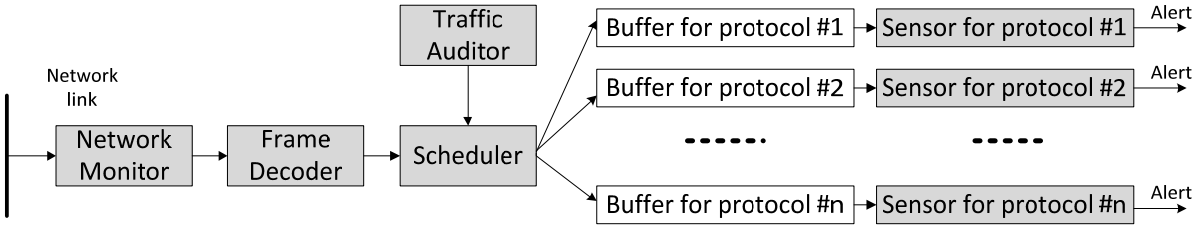


Figure 26: The architecture of the NIDS model by Zhuojun Zhuang.[56]

Splitting the traffic load into different detection sensors within the network is the primary idea to parallelize Snort. M. Shoaib Alam also designed an adaptive load balancing architecture, which uses some policies enforced on the splitter by a management console to adjust the splitting policies for keeping the load disparity among the sensors reduced. [42] The policy tries to minimize the packet duplication rate during the system operation, which can minimize the percentage of the duplicated traffic. A packet flow structure is shown in Figure 27.

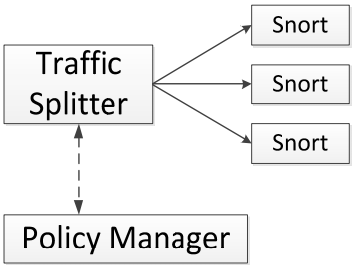


Figure 27: Adaptive Load Balancing Architecture of Snort by M. Shoaib Alam. [42]

Similarly, some research has been carried out to parallelize the inspection work load on different machines or sensors with one single network, which will distribute the candidate packets to a different machine. It usually needs a traffic classification in the centre machine to manage the jobs and a load balancing algorithm for the balancer to organise the work load.

Meharouech Sourour from Tunisia drew a structure to provide a high availability mechanism which is shown in Figure 28. It includes a classifier, a few balancers, some sensors for intrusion detection, a manager for output, and a database for alert records, logs and configured parameters. **[59]**

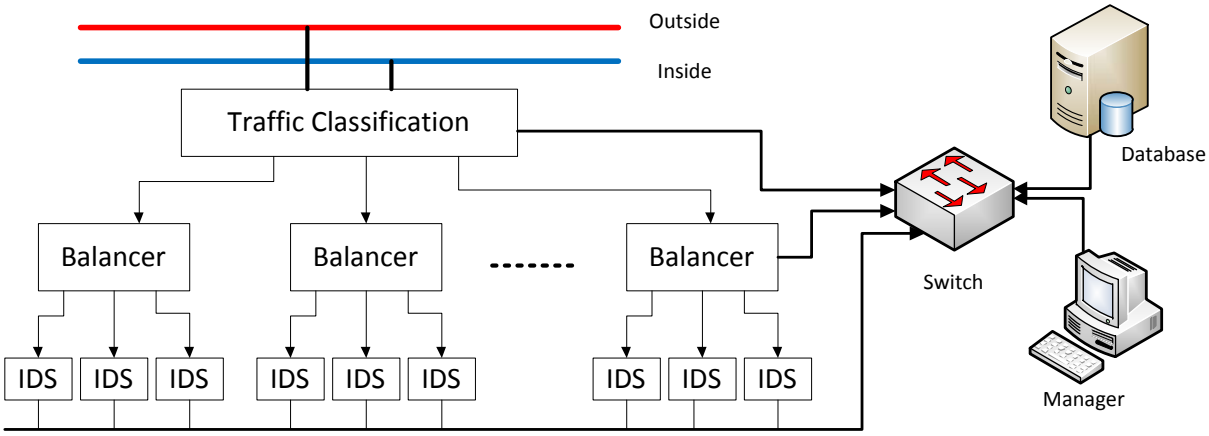


Figure 28: A Stateful Real Time Intrusion Detection System by M. Meharouech Sourour. [59]

Snort is a typical misuse detection which identifies evidence of malicious behaviour by matching it against predefined descriptions of attacks, or signatures. Compared with anomaly detection, which defines normal behaviour and attempts to identify any unacceptable deviation as possibly the result of an attack, misuse detection has a lower false alarm rate and a high miss detection rate. Some research has been carried out to reduce its miss detection rate. The main idea of this research is adding some anomaly detection features into Snort. For example, Lih-Chyau Wuu [44] from Taiwan gave Snort the ability to catch new patterns automatically and detect sequential attack behaviours by introducing an "Intrusion Pattern Discovery Module". This module can find single intrusion patterns and sequential intrusion patterns from a collection of attack packets in off-line training phases. It includes some data mining skills and concludes some new signatures from large stories of packets and then converts them into Snort detection rules for the following on-line detection.

2.5 Summary

As a popular open source NIDS/NIPS solution, Snort is widely used by many researchers in their parallelization of deep packets inspection and intrusion detection. However, as many researchers choose to use FPGA as the development platform, a small amount of research has been carried out based on an optimized network processor. At this point, although Snort officially works on a general CPU, almost no research has been carried out on the topic of Snort parallelization on a multi-core/ multi-processor computer.

3 Snort Code Analysis

3.1 Snort Overall Structure

Besides the external packet capture module (Libpcap in Linux and Winpacp in Windows), Snort includes four primary processing modules: packet decoder, preprocessor, detection engine and output module which are shown in Figure 29.

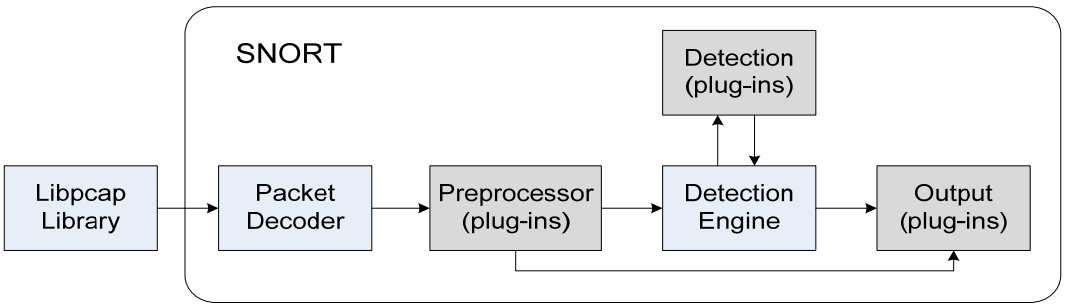


Figure 29: The Snort primary processing module. [10]

Derek L. Schuff evaluated a percentage workload on each primary module in the flow of packet processing, which is shown in Figure 30.[9] It shows the decoder and outputs only need a very small CPU resource; the detection engine requires about an 80% CPU resource.

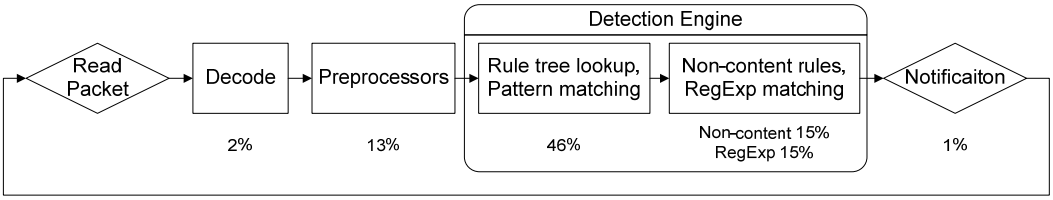


Figure 30: The Snort packet processing loop with the percentage of time spent in each phase. [9]

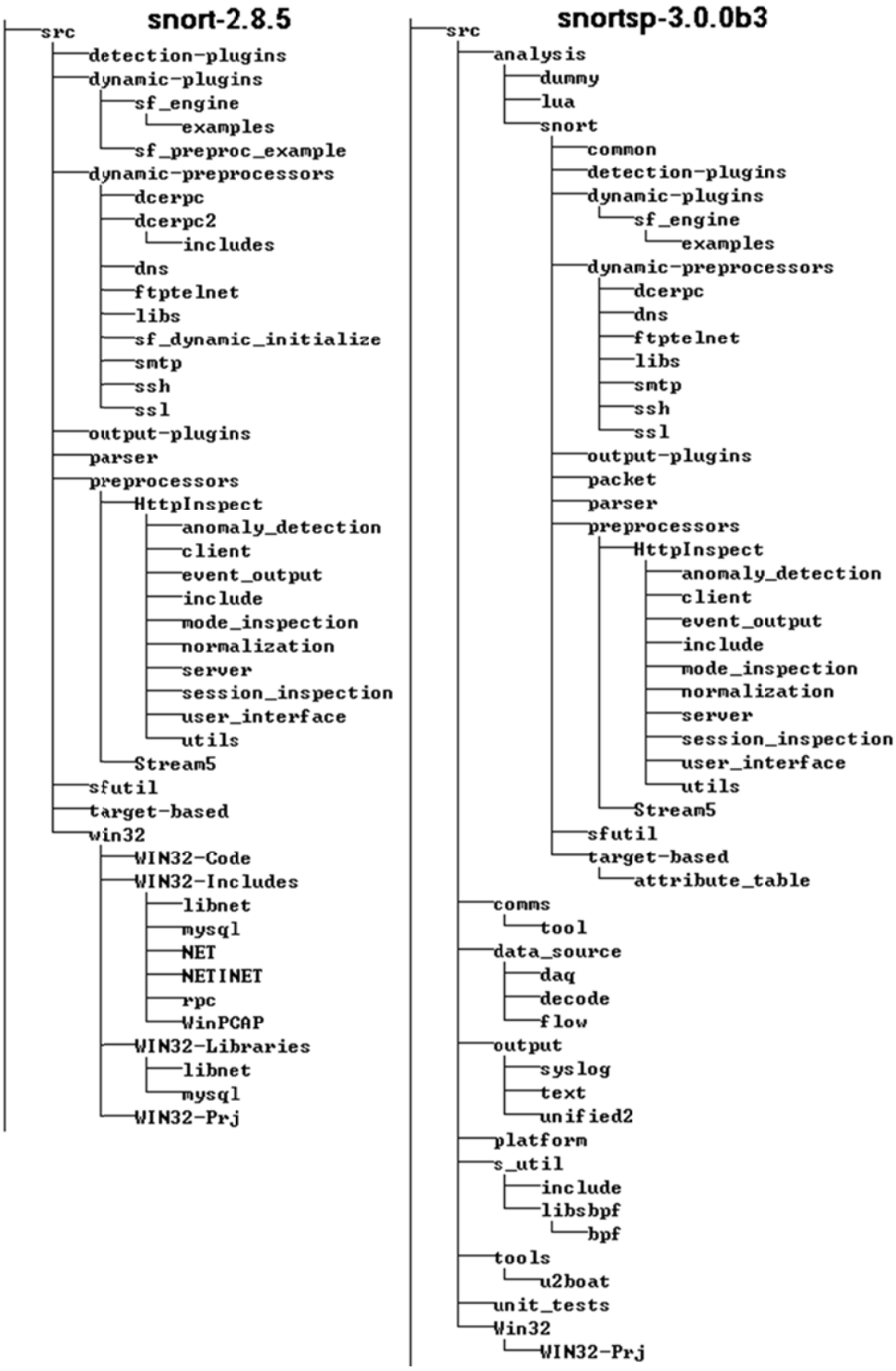


Figure 31: The code structure of Snort for version 2.8 and 3.0beta.

Now, version 2.8 is the latest stable version of Snort; and version 3.0 is the next generation of the Snort Security Platform, which contains the version 2.8 as its traffic analysis module, and some extra features to control the data flow. A code structure comparison of these two versions is shown in Figure 31. The version 2.8 Snort is located in the "/src/analysis/snort/"

folder of version 3.0 Snort Security Platform without much difference. This thesis will focus on version 2.8.

In version 2.8, most primary modules are in the "src" folder, such as "snort" - the main program entry, "decode" - decoder, "detect" – preprocessor and detect control, "fpdetect" – dynamic detection engine, "event" – output and so on. The folder "dynamic-plugins" contains all common functions of dynamic plugins for both preprocessor and detection and the folder "detection-plugins" contains all detection plugins. The folders "preprocessors" and "dynamic-preprocessors" contain both static and dynamic preprocessor plugins. All plugins are designed by different authors. The other folders contain different utilities. The real detection engines (different string matching engines) are in the "sfutil" folder. "snort.c" and "detect.c" are the two files containing all primary functions of Snort.

3.2 Snort Variables

There are two important global variables in Snort: "Packet" in "decode.h" and "SnortConfig" in "snort.h". "Packet" is the most important as it contains all the packet information for a single packet. There are three main types of information: (1) a pointer points to an address in the originally captured packet in the memory. The packet payload data will be used for the final detection. However, there will be some issues in the multiple processes program, because the pointed address in different processes points to different data, the same packet passing from one process to another does not pass the correct data by default. (2) The decoded information from the original packet, especially from the header, for example, the source and destination IP address, protocol and port number. (3) Some flags will be used for future packet detection, such as "preprocessor_bits" which identify which preprocessor will be used for the packet.

```
Typedef struct _Packet
{
    const struct pcap_pkthdr *pkth; /* BPF data */
    const uint8_t *pkt; /* base pointer to the raw packet data */
    … …
    const EtherHdr *eh; /* standard TCP/IP/Ethernet/ARP headers */
    const VlanTagHdr *vh; /* 802.1q vlan header */
    const uint8_t *data;          /* packet payload pointer */
    const uint8_t *ip_data;       /* IP payload pointer */
    … …
    IP4Hdr inner_ip4h; /* IP Header information, inc. IP Address) */
    IP6Hdr inner_ip6h; /* IP Header information, inc. IP Address) */
    IP4Hdr outer_ip4h; /* IP Header information, inc. IP Address) */
    IP6Hdr outer_ip6h; /* IP Header information, inc. IP Address) */
    uint16_t sp;                  /* source port (TCP/UDP) */
    uint16_t dp;                  /* dest port (TCP/UDP) */
    … …
    uint32_t preprocessor_bits; /* flags for preprocessors to check */
    uint32_t caplen;              /* captured packet length*/
    uint32_t http_pipeline_count; /* Counter for HTTP pipelined requests */
    uint32_t packet_flags;        /* special flags for the packet */
    uint32_t proto_bits;          /* protocol number */
    uint16_t dsize;               /* packet payload size */
    uint16_t ip_dsize;            /* IP payload size */
    uint16_t frag_offset;         /* fragment offset number */
    uint16_t ip_frag_len;         /* fragment length */
    uint8_t frag_flag;            /* flag to indicate a fragmented packet */
    uint8_t uri_count;            /* number of URIs in this packet */
    uint8_t csum_flags;           /* checksum flags */
    … …
    uint16_t configPolicyId;      /* configPolicyId in SnortConfig */
}Packet
```

Figure 32: The "struct" of a Packet in Snort.

"SnortConfig" contains almost all Snort configurations and running information. It includes some flags such as "run_flags" which record the Snort running mode; some general information from the configuration file, such as the home network, and its network mask; and the pointers point to the configured plugins and rules. These variables request the memory by different functions through "malloc", and are distributed in random locations of the memory.

"DynamicLibInfo *dyn_engines" points to a list of dynamic detection engines, it is located in "/src/dynamic-plugins/sf_dynamic_engine.c" in the source folder, and in "/usr/local/lib/snort_dynamicengine/libsf_engine.so" in a installed Snort system in Linux; "DynamicLibInfo *dyn_rules" points to dynamic detection libraries, it is located in "/src/dynamic-plugins/sf_engine/sf_snort_plugin_api.c" in the source folder, and in the

“/usr/local/lib/snort_dynamicrule/” directory in an installed Snort system in Linux; “DynamicLibInfo *dyn_preprocs” points to dynamic preprocessors, it is located in the “/src/dynamic-preprocessors/” folder in the source folder, and in the “/usr/local/lib/snort_dynamicpreprocessor/” directory in a installed Snort system in Linux. All of them can be initialled in two ways: (1) by the function “ParseCmdLine” in “snort.c” if the configuration is set in command line mode; (2) by the function “ParseSnortConf” in “snort.c”, if the configuration is loaded from the configuration file (normally “snort.conf” in /etc/ folder). All the three variables are loaded by function “LoadDynamicPlugins” in “snort.c”.

“OutputConfig *output_configs” points to the list of output plugins. “OutputConfig *rule_type_output_configs” defines the rule types which will log to the output modules, such as “tcpdump”, “syslog” and “mysql”

“RuleState *rule_state_list” points to a list of rule states. The rule is identified by gid and uid. In the rule state, it records if a rule has been enabled or disabled; and the action of the rule is either "alert", "drop", "sdrop", "log" or others. The rule state will be imported during the Snort initialization from the configuration files.

“ClassType *classifications” define the predefined classification list of Snort rules. The location of the classification file is defined in the “/etc/snort.conf” file, which usually uses “/etc/classification.config”. This classification list includes a classification name, a short description, and the priority of this type of rule. All Snort rules define its classification group by “classtype”. Snort imports this classification list in the function “ParseConfig()” by function “ConfigClassification()”, which is a part of “static const ConfigFunc config_opts[]”. The rule classification will be used in function “ParseRuleOptions” to group the Snort rules. The Snort rules will be parsed in the function “ParseConfigFile()” which is at the Snort initialization or after Snort rules change.

```
Typedef struct _SnortConfig
{
    int run_flags;
    uint32_t homenet;     /* record home network from configuration */
    uint32_t netmask;     /* record home network mask */
    uint8_t ignore_ports[UINT16_MAX];     /* config ignore_ports */
    int pcre_match_limit;                 /* config pcre_match_limit */
    char *dynamic_rules_path;    /* --dump-dynamic-rules */
    DynamicLibInfo *dyn_engines;     /* point to dynamic detection engine */
    DynamicLibInfo *dyn_rules;       /* point to dynamic detection rules */
    DynamicLibInfo *dyn_preprocs;    /* point to dynamic preprocessors */
    OutputConfig *output_configs;    /* list of output plugins */
    OutputConfig *rule_type_output_configs;  /* list of rule type output */
    SFGHASH *config_table;       /* table of config keywords and arguments */
    RuleState *rule_state_list;      /* point to the list of rule state */
    ClassType *classifications;      /* the classification list of snort rules */
    ReferenceSystemNode *references;  /* the reference list */
    SFGHASH *so_rule_otn_map;
    SFGHASH *otn_map;
    FastPatternConfig *fast_pattern_config;
    EventQueueConfig *event_queue_config;
    PreprocPostConfigFuncNode *preproc_post_config_funcs;
    PreprocCheckConfigFuncNode *preproc_config_check_funcs;
    ThresholdConfig *threshold_config;
    RateFilterConfig *rate_filter_config;
    DetectionFilterConfig *detection_filter_config;
    SF_EVENTQ *event_queue;
    SF_LIST **ip_proto_only_lists;
    uint8_t ip_proto_array[NUM_IP_PROTOS];
    int num_rule_types;
    RuleListNode *rule_lists;
    ListHead Alert;          /* Alert Block Header */
    ListHead Log;            /* Log Block Header */
    ListHead Pass;           /* Pass Block Header */
    ListHead Activation;     /* Activation Block Header */
    ListHead Dynamic;        /* Dynamic Block Header */
    ListHead Drop;
    PluginSignalFuncNode *plugin_post_config_funcs;
    OTNX_MATCH_DATA *omd;
    rule_port_tables_t *port_tables;
    PORT_RULE_MAP *prmIpRTNX;
    PORT_RULE_MAP *prmTcpRTNX;
    PORT_RULE_MAP *prmUdpRTNX;
    PORT_RULE_MAP *prmIcmpRTNX;
    srmm_table_t *srmmTable;   /* srvc rule map master table */
    srmm_table_t *spgmmTable;  /* srvc port_group map master table */
    sopg_table_t *sopgTable;   /* service-oridnal to port_group table */
    SFXHASH *detection_option_hash_table;
    SFXHASH *detection_option_tree_hash_table;
    tSfPolicyConfig *policy_config;
    SnortPolicy **targeted_policies;
    unsigned int num_policies_allocated;
} SnortConfig
```

Figure 33: The struct of SnortConfig in Snort.

Similarly to the classifications, “ReferenceSystemNode *references” records the reference list from “/etc/reference.config”.

IDS and intruders are competitors. When the IDS studies and analyses the intruders' technique and to get a new signature, the intruders are also studying the signatures of the IDS to avoid their detection. SO rules are designed to reduce the possibility, they are a kind of pre-compiled rule, and cannot be learned directly without a reverse engineering by intruders. "so_rule_otn_map" is used to store SO rules. And "otn_map" is used to store the normal rules.

3.3 Snort Initialisation

In Snort, function "Main()" in "snort.c" is the entry of the program. It will check if Snort is running as a server in the Windows system. If so, it will pass the process to "SnortServiceMain()" in "snort.c", then call "SnortServiceStart()" in "win32_service.c" to perform a group of special initialization for the Windows system, and return to "SnortMain()" in "snort.c"; otherwise, it will return to "SnortMain()" directly. "SnortMain()" will proceed with a large amount of initialization jobs before the packet processing loop. Figure 34 shows a flowchart of the entry process of Snort.

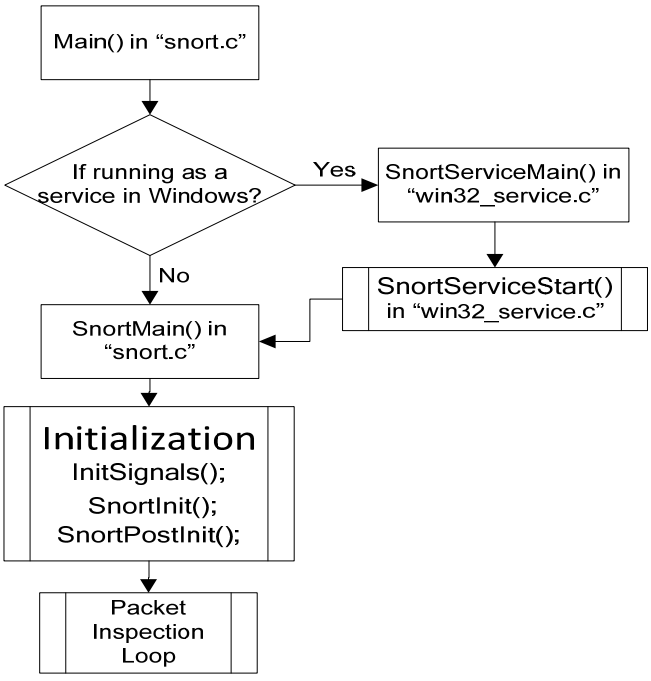


Figure 34: The entry process of Snort.

In function “SnortMain()”, Snort will initialise all signals; do a first stage initialisation in “SnortInit()”; initialise PCAP and open the network interface if the packets are directly read from an NIC; perform a post stage initialisation “SnortPostInit()“ and finally bring the system in to the working session “SnortProcess()”. Snort supports both transparent detection (NIDS) and inline detection (NIPS) mode. Here, an extra thread will be created for inline fail-open mode.

In function “SnortInit()”, all configured parameters will be imported; the signature rule will be parsed; and all enabled output plugins, pre-processors and detection plugins will be registered and initialised.

In function “SnortPostInit()”, the system will set up the mode of the network interface and carry out some extra configuration of different plugins.

3.4 Snort Detection Process

Function “SnortProcess()” is the entry to the processing section. In the function, another function “InterfaceThread()” is called to create a thread per interface. This interface is usually an NIC interface for a single input file. Currently, Snort will officially create the multiple threads for multiple interfaces, each interface will use a single thread, whether there are any traffic data in that interface or the traffic is heavy.

Function “InterfaceThread()” will call PCAP to capture or read a single packet one by one in a “while (1)” loop. A structure of the function is shown in Figure 35.

Snort uses “pcap_dispatch” or “pcap_loop()” to get the packets and directly send them to a call-back function “PcapProcessPacket()”. The working mechanism of LibPCAP will be discussed in the following section.

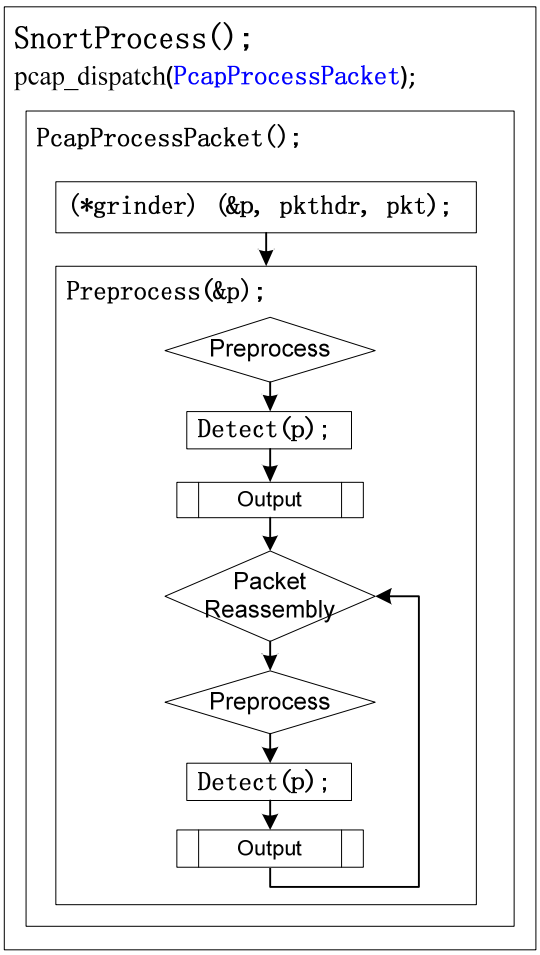


Figure 35: The structure of the process functions of Snort.

Function "PcapProcessPacket()" contains the whole job of processing a single packet. It uses a function pointer "grinder" to perform decoding corresponding to different types of packets.

When PCAP reads a packet, the read packet is stored in two different variables: "const struct pcap_pkthdr * pkthdr" and "const u_char * pkt". "pkthdr" has the packet header, size and time information; and the whole packet payload is stored in "pkt". The decoded packet is stored in a local variable "Packet * p". However, the decoded information does not copy the packet payload to the local, it only uses a pointer to point the location in the variable "pkt". Therefore, all the above three variables are necessary for the later processing.

The function "preprocess(&p)" will not only carry out the pre-processing job but firstly passes the decoded packet "p" through all pre-processors, and then performs detection once,

sending the output if there is any. After the first loop detection, some fragment packets will be marked in the variable “p” to be reassembled by predefined reassembly functions. Then it passes the reassembled packet through all pre-processors and performs the detection again. After that, the newly detected packet will be checked for fragment until there is no fragment in the packet. However, if the normal traffic only contains a very small number of fragment packets, most packets do not need to be reassembled and detected more than once. There are also a very small number of packets, which need to be reassembled a few times.

In the IDS mode, the packet “p”, its header “pkthdr”, and its payload “pkt” will be directly dropped at the end of detection. In the inline (IPS) mode, the packet will be resent if nothing has been detected.

3.5 Snort Signature Rules

As a misuse detection system, Snort uses signatures to identify attacks. When a packet passes IDS, the detection engine will try to match all signature rules with the packet in order to identify the intrusion action. If a new type of intrusion is not in the signature database, it will not be found by the IDS. Basically, there are two types of rules that can be downloaded by the Snort website: subscriber rules and community rules. Subscriber rules contain all the newest rules from official releases, and are only available to the paid subscribers; community rules are free to download and have fewer rules than subscriber rules. Currently, the community rule set contains over 7000 rules in 55 groups. All signatures are stored in a group of “*.rules” files in the “rules” folder. An example of a Snort rule is shown in Figure 36. In the example, it defines a packet sent from any IP to any IP with defined HTTP servers, and TCP port number (usually 80 or 443), if a keyword “LOCK” is found in the payload with the first 5 bytes, then Snort will send an alert with a message “WEB-IIS WebDAV file lock attempt”. “Offset” is another important keyword in the rule force Snort inspects the packet payload from a number

of bytes. In addition, the content defined in the Snort rule is case sensitive, case insensitivity content will be defined by regular expression.

```
alert tcp any any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-IIS WebDAV file lock attempt"; content:"LOCK "; depth:5;)
```

Figure 36: An example of a Snort rule.

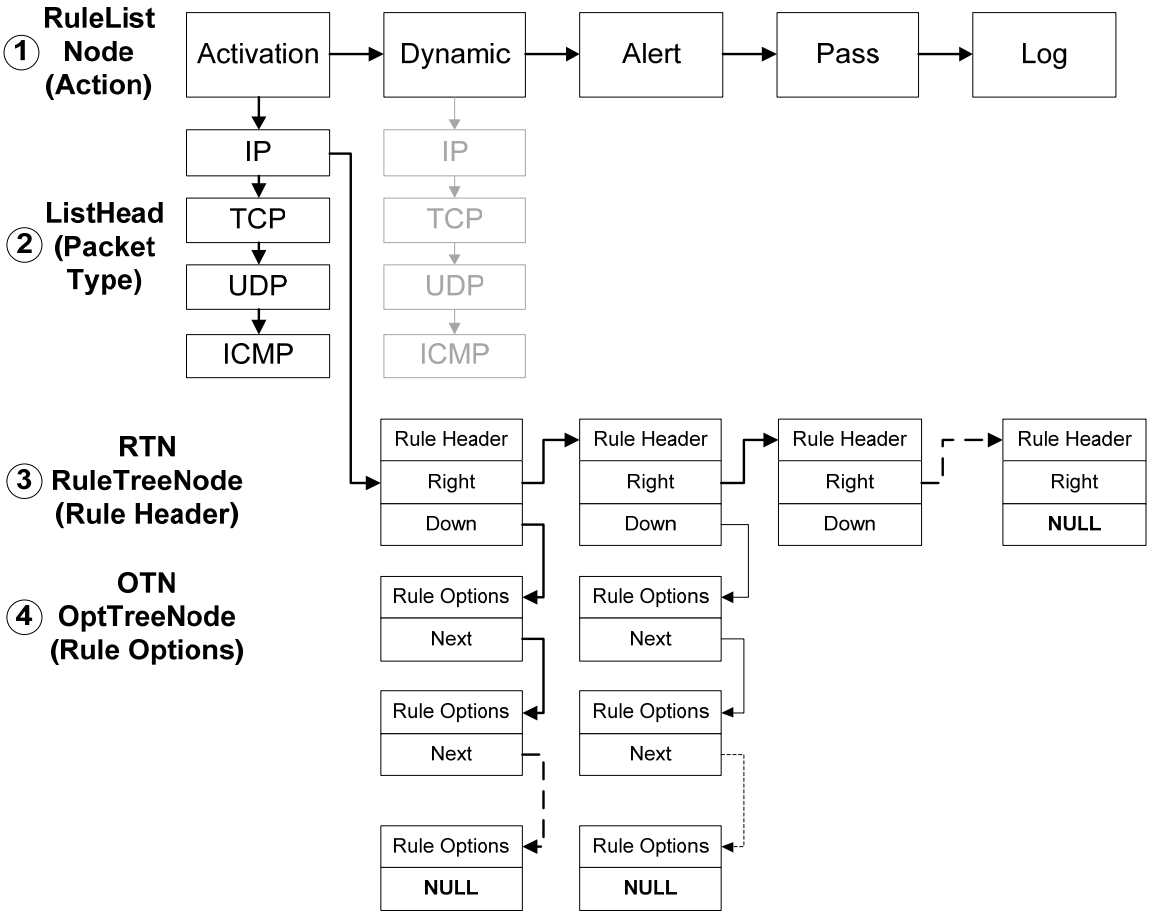


Figure 37: The structure of theThree-Dimension Chain, 1st Classification Rule Tree.

In order to perform inspection, all rules need to be imported from the files into the memory with a high performance structure. Snort uses a "three-dimension chain" to organise the rules in the memory. This concept classifies the rules by action, packet type and packet header into a rule tree. Figure 37 shows the structure of the rule tree. On the top (list) level, Snort has five different actions: activation, dynamic, alert, pass and log. On the second (packet type) level, Snort only supports four types of packets: IP, TCP, UDP and ICMP. Snort defines "Rule Tree

Node (RTN)" and "Option Tree Node (OTN)" relating to the header and options of rules (Figure 38) owing to the fact that many rules have exactly the same header.

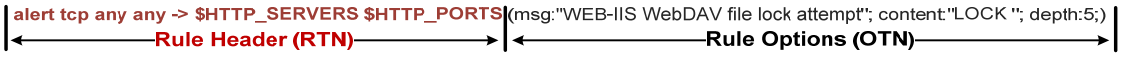


Figure 38: The division of an example rule in RTN and OTN.

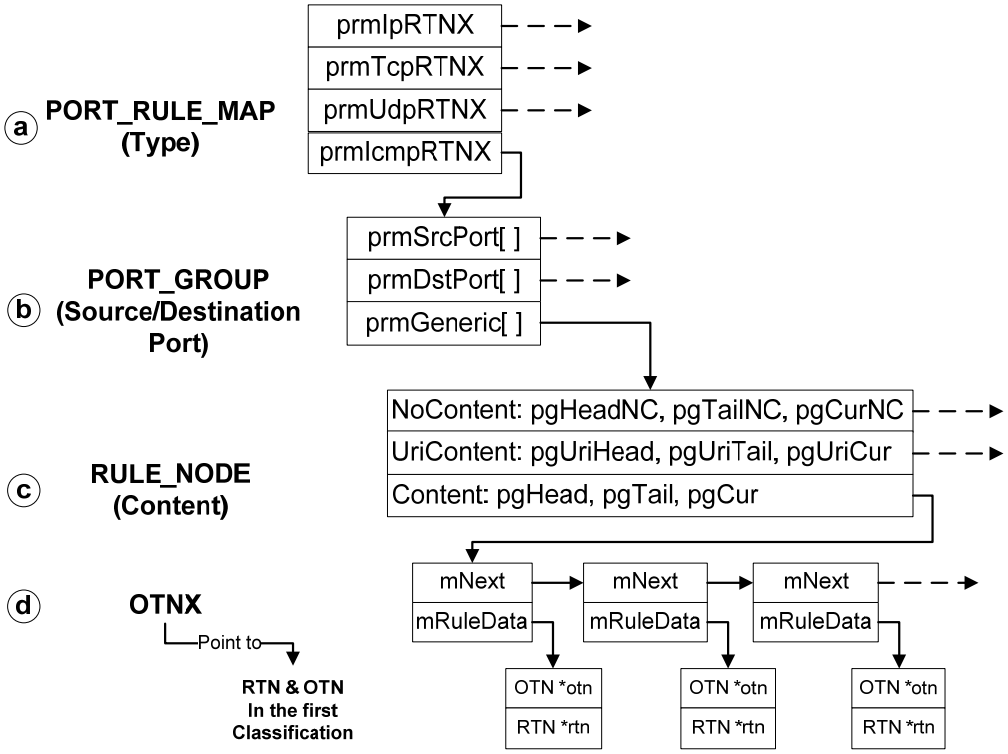


Figure 39: The structure of the Fast Packet Detection Chain, 2nd Classification Rule Tree. [14]

Function CreateDefaultRules() is used to create the top level RuleListNode by calling function CreateRuleType() five times to create "Activation", "Dynamic", "Alert", "Pass" and "Log" RuleListNode. ListHead is defined in CreateRuleType(). Then Snort uses function ParseRulesFile() to analyze and read the rule file line by line, and pass the reading to function ParseRule(), which will parse each reading rule and search for lines that are not detection rules. If a rule is a detection rule, it will verify it and write it into memory by using function ProcessHeadNode(). ProcessHeadNode() fills RTN, and then calls function ParseRuleOptions() to create OTNs.**[60]**

Owing to the large number of rules, Snort uses a secondary rule classification to reduce the number of rules in a single class. In the classification, each ListHead is classified again according to the source/destination port (RTN) and its content (OTN). Four new groups are added between the second level (ListHead) and the third level (RTN) in the first classification rule tree: PORT_RULE_MAP (a modified structure of packet type), PORT_GROUP (stores the source and destination port), RULE_NODE (includes three different contents: content uri-content and no-content) and OTNX (points to the address of RTN and OTN which is defined in the first classification). A structure is shown in Figure 41. This chain is used by the "Fast Packet Detection" engine, which can greatly speed up the detection performance.

Fast Packet Detection is initialized through function fpCreateFastPacketDetection() in "./fpcreate.c" which will go through all rules in the first classification rule tree using RuleLists (a RuleListNode pointer). Each rule is classified according to its content (Content, UriContent or NoContent). Content is determined through the OTN associated with the rule. Then function prmAddRuleXX() (prmAddRule(), prmAddRuleUri() or prmAddRuleNC()) is used to determine if the rule is bidirectional and sorts rules in tables according to source and destination port. There are three groups of PORT_GROUP: one is the source port table (prmSrcPort), another is the destination port table (prmDstPort) and the last is the generic table (prmGeneric) which is used for rules with "srcport = any" and "dstport = any". However, a packet may be in both prmDstPort and prmGeneric groups. Therefore, function prmCompileGroups() is used to link the prmGeneric to the prmDstPort for that port. For example, a rule belongs to the "TCP prmDstPort 21" and prmGeneric; the detection engine will go through the prmGeneric group first, and then go through the "prmDstPort 21" group. [60]

3.6 Snort Plugins

Plug-in is an important concept of Snort. At this point, there are three types of plugins in Snort: output plugins (in the "output-plugins" folder), pre-processors (in the "preprocessors" folder), and detection plugins (in the "detection-plugins" folder).

Output plugins include some kind of passive log systems and such plugins including "LogTcpdump" plugin can output the log in the TCPDump format; "AlertSyslog" plugin can send an alert to a syslog server; "Database" plugin can output an alert to TransSQL database, such as MySQL and MS SQL.; "AlertFast" plugin can simplify the output alert; "AlertFull" plugin can generate a detail alert; "AlertCSV" plugin can output an alert in a standard CVS format, and so on.

Detection plugins do not only include the pattern matching engines, but also include some special string operations such as the length for size detection of a string.

The pre-processor is a very important part of packet detection. Without pre-processors, the simple pattern matching module finds it hard to find all matching from unformatted packets. Snort consists of some general pre-processors, such as "Frag2" for packets defragment/reassembly, "Stream4" for session reorganization, and some special purposes pre-processors, such as "HttpInspect" for decoding and formatting http URL, "ARPspoof" for ARP spoof recognition, "Bo" for BO backdoor recognition, "TelNeg" for telnet session formatting, and so on.

The work of a plug-in includes three steps: register, initialization and action. Therefore, each plug-in has three functions: the register function (RegA, RegB, …), the initialization function (InitA, InitB, …) and the action function (ActionA, ActionB, …). The register will initialize all plug-ins which are supported by Snort and make a plug-in initialization chain which will

encapsulate all plug-ins' initialization functions (Figure 40). Then initialization will read the configuration file "snort.conf", and get the plug-in which has been enabled by the user. By using the plug-in initialization chain, it builds a new plug-in action chain which includes all enabled plug-ins (Figure 41). Finally, the action executes the action functions one by one following the plug-in action chain.

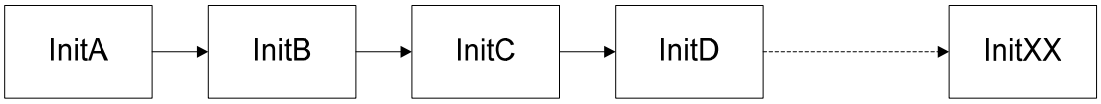


Figure 40: plug-in initialization chain created by register.

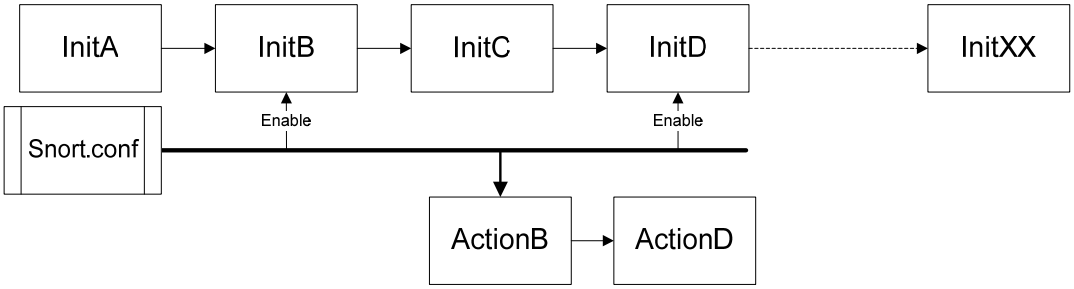


Figure 41: Plug-in action chain created by initialization from the plug-in initialization chain.

In the detection process, when Snort needs to call these enabled plugins, it only calls the first plugin in the chain, and goes through all the others like an automaton. There is one stop in the middle of the plugins.

3.7 Packet defragment "frag2"

The network hardware has a maximum transmission unit (MTU). It is the largest protocol data unit in bytes that can be passed on through the data link layer of a hardware interface. The value is usually defined in the hardware, and cannot be modified by the operation system. In the Ethernet 802.3, the MTU is 1492; the Ethernet v2, which is almost always used by all network devices, has a MTU of 1500; and some modern network devices now support "Jumbo Frames" which have a MTU up to 9000. The maximum size of an IP packet is 0xFFFF (65536 bytes). When a packet bigger than the MTU is sent by OS, the hardware will automatically fracture the big packet into multiple fragment packets. Each fragment packet

will have its own IP header. Therefore, attackers can construct some special fragment packets to avoid detection. An example of these special fragment packets is when the whole of the attacking codes are divided and distributed in different fragment packets.

Snort will use the preprocessor“frag2” to defrag all fragment packets to the original IP packets, and then pass the original packet into the detection engine for final inspection. As Snort processes per packet, and “frag2” is needed to handle multiple packets, it changes the original Snort packet processing structure to become a compulsory plugin.

When a number of continuing packets come into Snort, they may belong to a single IP packet; but they may also belong to multiple different packets. Therefore, the system requires an array of at least two dimensions to store these packets. Snort uses two levels of binary trees to store packet information. The primary binary tree keeps the packet information, such as IP, protocol, fragment flags, fragment bytes, original packets size, fragment packet numbers, and so on. Each node (struct ubi_btNode Node) in the primary binary tree will point to a struct “FragTracker”. “FragTracker” contains the nodes (struct ubi_btRoot fraglist) and the point (struct ubi_trRootPtr fraglistPtr) of the secondary binary tree. Each node of the secondary binary tree will point to a “Frag2Frag” struct, which contains the data, size and offsets of a fragment packet.

For a new incoming packet, “frag2” will compare its features to see if they belong to the features of the primary binary tree. If yes, it should be a new fragment packet of this node, and adds it to the secondary binary tree. If not, it adds it as a new node. A defragmenter flow chart is shown in Figure 42.

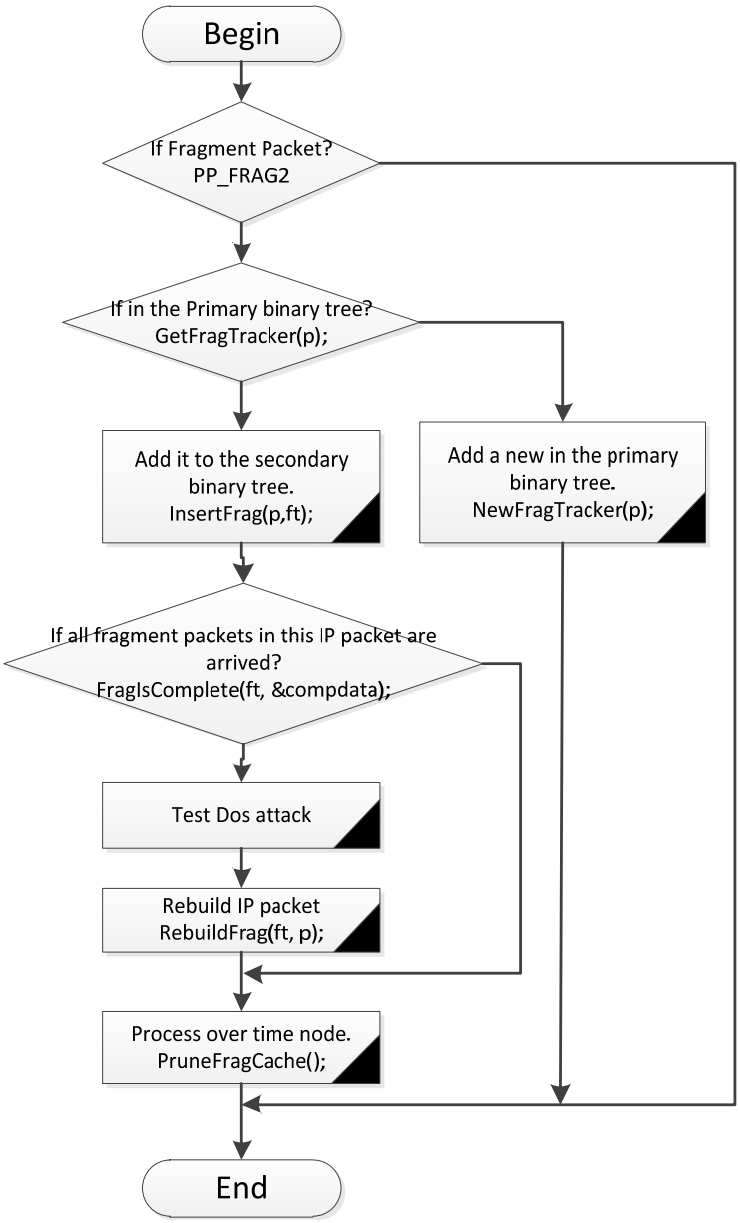


Figure 42: The defragment process of Snort.

3.8 Detection Engine

Function "Detect(p)" is the entry of the Snort detection process, which is called the "preprocessor(p)" function. In the "Detect(p)" function, it imports the rule list and calls the real detection module "fpEvalPacket(p)".

Function "fpEvalPacket(p)" is the interface of the detection engine and it will identify the type of packet coming into the detection engine as TCP, UDP ICMP or IP. It then calls the corresponding function "fpEvalHeaderXXX", such as "fpEvalHeaderTCP",

"fpEvalHeaderUDP", "fpEvalHeaderICMP", and "fpEvalHeaderIP". All four functions are very similar.

```
Typedef struct {
    PORT_GROUP * PG;    // point to the corresponding PORT_GROUP
    Packet * p;         // the candidate packet.
    int check_ports; // if need to check port number, if TCP or
UDP?
    int iMatchInfo ArraySize; // the number of type, default is 5.
    MATCH_INFO *matchInfo;  // All found Match, point to
OTNX(rules).
} OTNX_MATCH_DATA;

OTNX_MATCH_DATA *omd = snort_conf->omd; // declared in function
                                        // fpEvalPacket(Packet *p);

Typedef struct {
    OTNX *MatchArray[MAX_EVENT_MATCH];
    int iMatchCount;
    int iMatchIndex;
    int iMatchMaxLen;
} MATCH_INFO;
```

Figure 43: The struct of "OTNX_MATCH_DATA" in "InitMatchInfo" function.

TCP is the most common protocol in the network. In the function "fpEvalHeaderTCP", it will firstly call the "prmFindRuleGroupTcp" function to find the corresponding "PORT_GROUP" under RTNX based on the port information of the packet. Then the function "InitMatchInfo(&omd)" is used to prepare all related information of the "PORT_GROUP", and initialize struct "OTNX_MATCH_DATA omd" which is used to store all detection results. The structure of "OTNX_MATCH_DATA" is shown in Figure 43.

After the initialization, function "fpEvalHeaderSW" will be called to organise the later packet matching. The "PORT_GROUP" includes three types of nodes: content, uri-content, and no-content. A packet is required to carry out all three types of matching.

"Content" is a general group of characters. "uri" stands for the "Universal Resource Identifier" on the web and "uri-content" is usually the address of a webpage or FTP route, e.g. "http://www.snort.org/assets/166/snort_manual.pdf". A single packet may include more than

one URI. If any URI has been found by the pre-processor “http_decode”, the packet will be required to perform uri-content matching. Both content and uri-content matching will use the multiple pattern search engine “mpseSearch()” to perform pattern matching. And “mpseSearch()” will use one of the user defined pattern matching algorithms to perform pattern matching, such as Basic Keyword Search Trie, Aho-Corasick DFA, Aho-Corasick NDFA,or Aho-Corasick NFA. The type of pattern matching algorithm is saved in “p->method = SnortConfig sc->fast_pattern_config->search_method”. The Aho-Corasick pattern matching algorithm is the default algorithm of Snort.

Table 2: Typical Snort Detection Plugins

RTN Detection Plugins (call by fpEvalRTN()):	
Check Bidirectional	Check if the rule header has a bidirection character “<>”.
CheckSrcPortNotEq	Check if the source port of the packet does not equal the rules.
CheckDstPortNotEq	Check if the destination port of the packet does not equal the rules.
CheckSrcPortEqual	Check if the source port of the packet equals the rules.
CheckDstPortEqual	Check if the destination port of the packet equals the rules.
CheckSrcIP	Check the source IP.
CheckDstIP	Check the destination IP.
OTN Detection Plugins (call by fpEvalOTN ()):	
dsize	The length of load.
Session	Get the user data in the TCP session.
ICMP	All ICMP attacks related detection tools
resp	The dynamic responding detection tools

"No-content" matching includes all other matchings in the Snort rule keyword options. Snort needs to run over all nodes in the "PORT_GROUP" one by one to perform the matching. Functions "fpEvalOTN()" and "fpEvalRTN()" are used for no-content matching. They will match the OTN and then the RTN by calling the corresponding detection plugins to perform the matching jobs. Some typical plugins are listed in Table 2.

When a matching has been found, the detection engine will call function "FpAddMatch()" to write the match into variable "OTNX_MATCH_DATA omd". The actual data storage location of "omd" is in the globe variable "snort_conf->omd". There is an array "MatchArray" in the "omd" store all matching the OTNX pointer.

4 LibPCAP Mechanism Analysis

Snort uses LibPCAP in Linux and WinPCAP in Windows to capture packets in the network or read packets from a saved pcap file. Libpcap is a hardware independent open source library that provides a high level interface to a network packet capture system.[61] Some Snort related features of LibPCAP will be analyzed in this session, and all analysis is based on version 1.0.0.

When network packets are sent and received by a network card, the network driver in the kernel space will grab/put from/to the protocol stack of the OS, and then the OS will pass the packet to the application in the user space. The data flow is bidirectional. For a high performance packet capture, half of LibPCAP is in the boundary of the kernel space, and grabs both transmitted packets and received packets from the driver. The other half is in the user space to process the packets. A structure is shown in Figure 44.

In general, the LibPCAP includes three parts: device initialization, BSD Packet Filter (BPF), and packet processing loop. A packet process structure is shown in Figure 45.

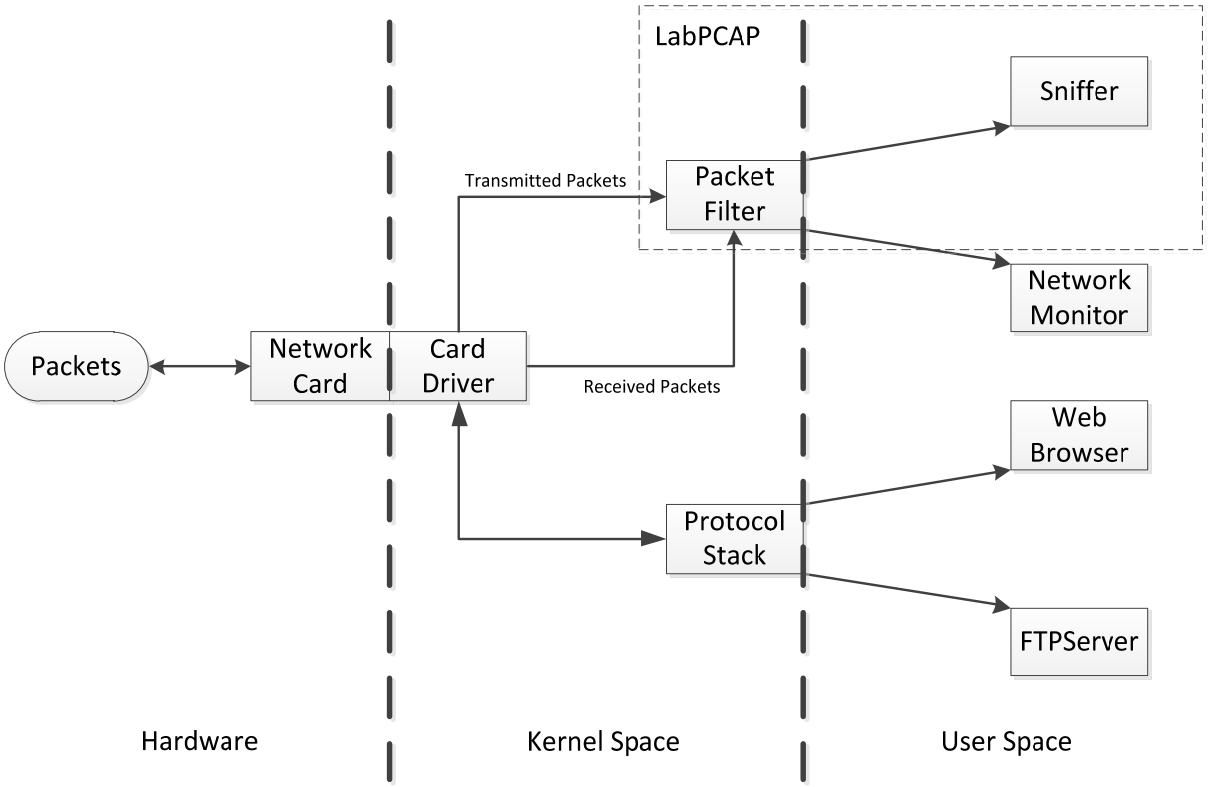


Figure 44: The structure of packets capture. [62]

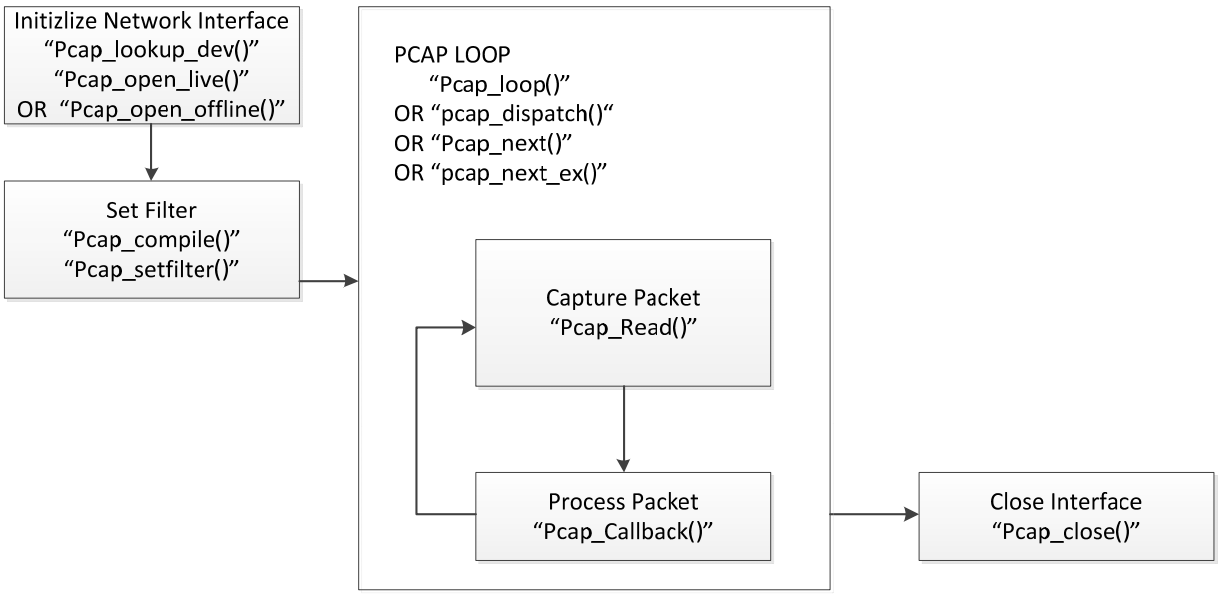


Figure 45: The packet process structure of LibPCAP.

The first step of using LibPCAP is device initialization. In the Linux system, all network devices are visual files. They are above the device driver, and below the network protocol layer. The real network adapters have a name eth0, eth1, eth2, and so on. LibPCAP supplies a device lookup function “Pcap_loopup_device()” to list all network devices and uses “getifaddrs()” to get their IP address and related information. Then all found devices are added to the “pcap_if” list by “add_addr_to_iflist()” and “add_or_find_if()”.

After all network devices are found, function “Pcap_open_live()” can be used to open the devices in the “pcap_if” list. Snort will open the network device configured in the file “snort.conf”. If nothing has been configured, it will try to open all devices in the “pcap_if” list. It then uses “InterfaceThread()” to create a thread for each device interface. In addition, Snort supports three socket types: “SOCK_PACKET”, “SOC_RAW” and “SOC_DGRAM”. “SOCK_PACKET” is a socket type of linux kernel 2.0 and an early one. It does not include some packet type information, and the returned packet length is the reading length, not the original packet length. “SOC_RAW” and “SOC_DGRAM” are new socket types from linux kernel 2.2. “SOC_RAW” will directly return the reading data from the data link layer without any modification. “SOC_DGRAM” will return the cooked packet, which will remove the data link layer header of the packets. By default, Snort will use “SOC_RAW” in promisc mode.

The most important feature of LibPCAP is the BSD Packet Filter (BPF) mechanism. It can filter out some unnecessary packets, or only permit some specific groups of packets to be read by the sniffer. Normally, when a packet comes into the network interface, the data link layer driver will deliver it to the system socket. With BPF, the driver will call BPF to filter the packet first, and then BPF puts it in its output buffer. Finally, the driver will take control of the packet again. Before using BPF, the filter configuration needs be compiled first by “pcap_compile()”, and then applied on BPF by “pcap_setfilter()”.

There are four functions which can be used to read packets: “pcap_loop()”, “pcap_dispatch()”, “pcap_next()”, and “pcap_next_ex()”.“pcap_loop()” will continue to read packets until an error; “pcap_dispatch()” can be used to read a number of packets and “pcap_next()” will call “pcap_dispatch()” to read one packet only. On the Linux system, “pcap_dispatch()” and “pcap_next()” are exactly the same, as “pcap_dispatch()” can only read one packet each time. Snort uses a “while(1)” loop with “pcap_dispatch()” to read one packet each time and uses an external callback function to process the packet. For Snort, the callback function is “PcapProcessPacket()”.

On the Linux system, all the live packet reading functions will call “pcap_read_packet()” to read a single packet. It will call the Linux system function “recvfrom()” to directly read packets from the socket, and copy the whole packet data into the location “bp + offset”, where “bp = handle->buffer + handle->offset”. “handle” is the pcap handler, and the memory of “handle->buffer” is allocated during the device initialization “pcap_activate_linux(pcap_t *handle)” for one packet only with a size “handle->bufsize + handle->offset”. The “offset” is the reserved space for the data link header and normally equals 16 bytes. After reading the packet, the data link header, such as the time of reading, size of capture and original packet, will be calculated and filled in “pcap_header”. Finally, the packet “bp” and its header “pcap_header” will be passed to the callback function.

“pcap_next_ex()” is a packet capture function without an external call back function. It uses a fake call back function “pcap_fakecallback()” instead of the external call back function. And the fake call function will pass the location of the captured packet, and its data link header to the external program through “userdata” variable. However, the actual data of the packet data is still in the location “bp”, to which the memory is allocated by LibPCAP.

III. IMPLEMENTATION AND RESULTS ANALYSIS

5 Pattern Matching

In the previous chapter, Figure 30 showed a work load between different modules of Snort. The work load of pattern matching is almost half of the work load of Snort and Snort is still using a 35 year old algorithm "Aho-Corasick" which is a single thread single process algorithm. There is a not very efficient multiple threads multiple processes based pattern matching algorithm at this point. This chapter will analyze some possibility of using some parallelized pattern matching algorithms to improve the performance of Snort.

The analysis is of deep packet inspection in future computing, and based on the following differences from the current computer system:

- A general purpose processor is used.
- The processor has a similar frequency to a current computer system.
- There are a large number of cores in the processor.
- The processor may have multiple threads (hyper-thread).
- The bus bandwidth of the processor is bigger.
- The size of the computer word will be bigger, e.g. 64 bit, 128 bit, or 256 bit
- It has a bigger and faster processor cache.
- The system has a bigger memory and the memory access speed is faster.

5.1 General parallelization methods

In general, a job can be parallelized by "function parallelization", "data parallelization" or "pipelining". For different jobs, if there is no dependency, they can be processed in parallel by "function parallelization". A large amount of data which has the same processing steps can

be processed in parallel by “data parallelization”. For example, the Bit-Parallelism algorithm is a type of “data parallelization” in a single processor. If a single job can be divided into multiple sections, and the work loads of each section are similar, then pipelining can be used for the parallelization.

From the pattern matching algorithms mentioned in the background section, it is easier to see there are not many non-dependent functions. Hence, function parallelization is not a good option.

Although pipelining is a direct way to perform the parallelization based on the serial string matching algorithm, however the job is already very small for each single comparison. When data is transmitted from one processor to another processor, it has to transfer the data into a shared memory. As the memory speed is more than 100 times slower than the cache speed, pipelining may take more time than a single process. Therefore, pipelining is not a good option, either.

“Data parallelization” may be an easy way to parallelize the candidate text in most types of pattern matching algorithms. Only the Brute Force algorithm has a totally independent data relationship, but it is an extremely slow algorithm. Otherwise, a simple idea is to divide a text string into multiple “p” substrings. An example based on four cores is shown in Figure 46. The size of the string which is required to be searched by each core is n/p+m, and n/p must be much bigger than m. Hence, a long pattern or a short text will decrease the performance. In the application of intrusion detection, the maximum length of an Ethernet packet is 1514 bytes; and the maximum of a jumbo frame is 9000 bytes. After decoding and packets payload it is relatively too small. In addition, the Bit-Parallelism algorithm is initially designed to perform data parallelization on a single processor. It is easy to be converted into a multi-processor platform.

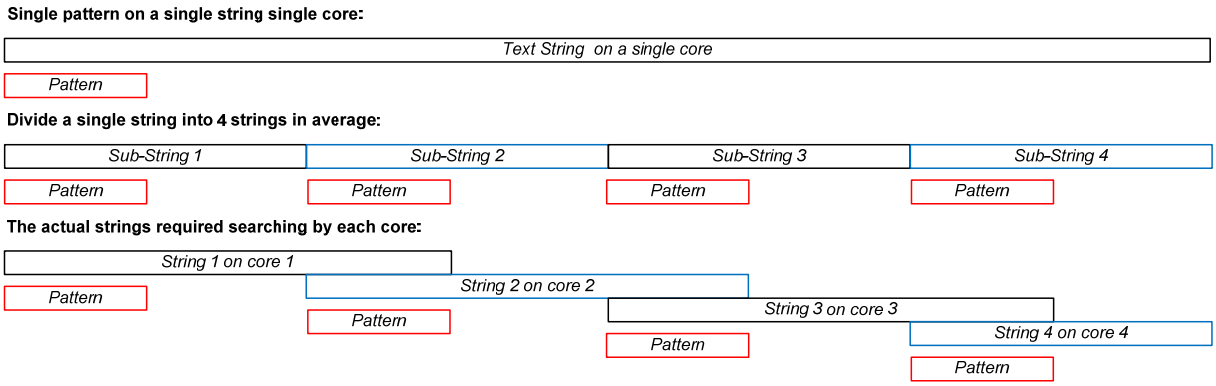


Figure 46: Data parallelization on Window shift algorithms on 4 processor cores.

In the later sections, some modified pattern matching algorithms will be described and analysed. As many multiple pattern matching algorithms are developed from a single pattern matching algorithm, some single pattern matching algorithms will also be described.

5.2 Reversed Bit-Parallelism Algorithm

Usually, a single character is encoded in 8 bits. However, the modem processors have 32 bits or even 64 bits, and the word can be increased to 128 bits or more in the future. When using a single character as a unit, it will waste most of the word width. The Multiple Bit-Parallelism algorithm described in the background can be used to fill out all computer words. When a single word is not enough, other processors can be used to supply the extra computer word. A structure example is shown in Figure 47.

In addition, the role of pattern and text can be reversed based on the bit-parallelism algorithm. Reversed Bit-Parallelism algorithm (RBP) will analyse the text in the pre-processor instead of the pattern, and use the known characteristics of the text to check if a pattern is in the text in the search engine. If the pattern exists somewhere in the text, a post-processor can quickly check its positions. Because the length of the text string is usually not short, it can be easily distributed into multiple processors by data parallelization and the ideal Bit-Parallelism algorithm.

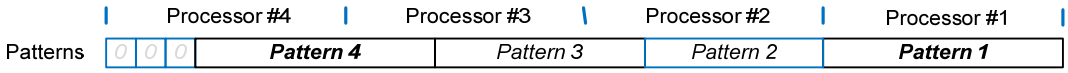


Figure 47: Multiple Bit-Parallelism algorithm for multicore computers.

For a pattern matching algorithm, some definitions are listed below:

- A string is defined as a one-dimensional finite sequence of symbols within an alphabet ∑ with a finite set of characters (σ), both pattern and text are strings.
- Let p be a pattern of length m, and T a text string of length n.
- A set of multiple patterns is presented by P = {p1, p2,…pr};
- A text T has multiple substrings {t1, t2,…tr} from left to right.
- The number of bits in a computer word (the word size or word length) is denoted by w.
- The number of cores/processors in a multi-core processor or distributed computing environment is denoted by c.
- Integer i (i <= (m-1)) will be used for the index of a pattern; and integer j (j <= (n-1)) will be used for the index of a text.
- Asterisk (*) denotes any (other) characters in the alphabet ∑.
- Symbol ε denotes an empty character.
- A string is written from left to right, the leftmost character is the first character; in the memory, a computer word will store from right to left and the first bit is stored at the rightmost position.

In the pre-processor, an S table is used to present the states (existing / not existing) of a character from the alphabet in the position of the text string. With the Reverse Shift-AND Algorithm, each character in the alphabet has a default state 0 (not existing). By calling over each character in the text, the states of each character in the text will set its position bit to "1" at the position of the text. In another word, the "1" in the S table indicates the character of the

S table is in the position of the text. After the pre-processor, each character in the alphabet has its own S table. Each S table has a size of n; and the total size of the S tables is (σ x n). The pre-processor has a time complexity O(n).

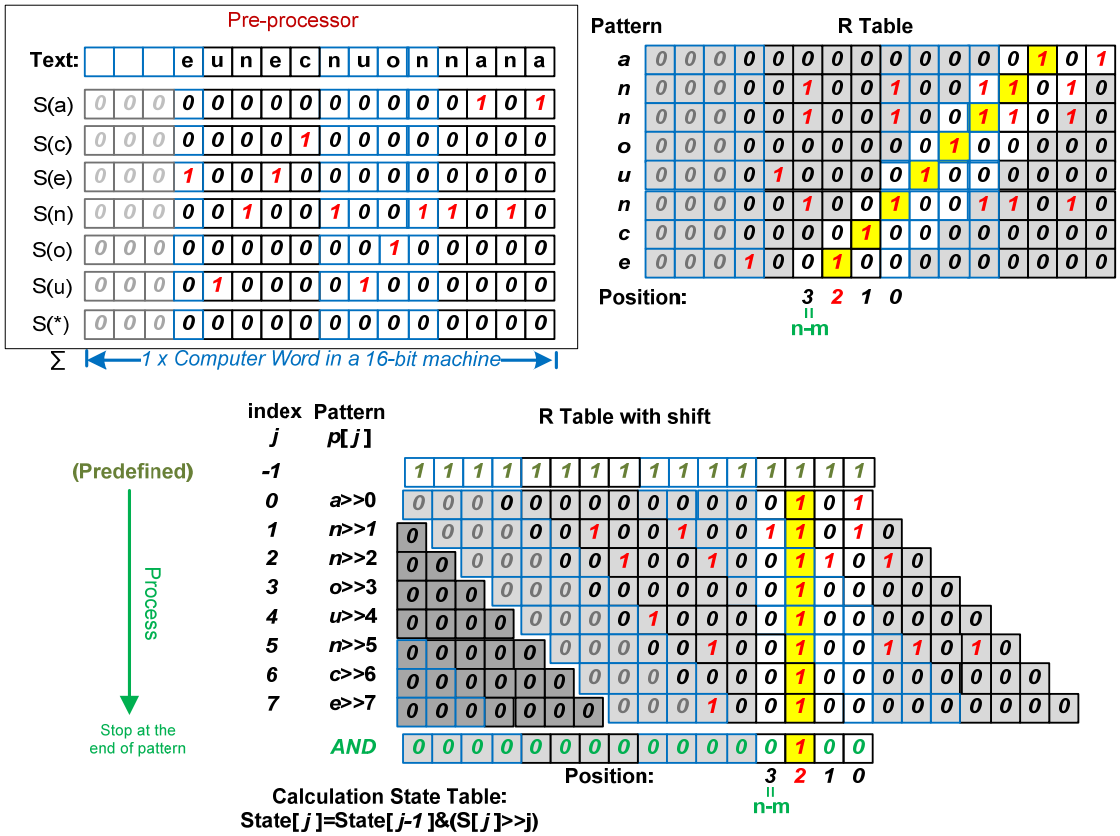


Figure 48: Search pattern "announce" in the text "anannouncenue" with one-core in a 16-bit computer by Reversed Bit-Parallelism (Reversed Shift-AND) Algorithm, n = 13, m = 10, c = 1, w = 16. Characters are encoded in ASCII. (Compare with Figure 7 to see the differences with BP)

By queuing the S table in the order of pattern, it is easy to identify a slope line with "1" from the first row to the last row if the whole pattern appears in the text, which is shown in the R table in Figure 48. By shifting i bit on each row (where i is the index of the pattern), the all "1" line can be queued in a single vertical column. By AND all rows in the R table after a shift, we can have a single word, the "1" bits in that word right to left indicate the position of the pattern in the text. If the pattern appears in the multiple position of the text, there will be more than one "1" bit. If there is not the pattern in the text, the whole word is 0. Because the position bit indicates the position of the first bit of the pattern, it can only be in the range of between position 0 and position (n-m). It is not necessary to consider the calculation outside

of this range, which has the grey background cells in the figure. An example of the RBP algorithm is shown in Figure 48.

In the implementation, it is not necessary to store the whole R table in the memory. A state word can be calculated when each character in the pattern has been read. The value of the initial state is 1w. The current state: State[j]=State[j-1]&(S[j]>>j). It is different with Baeza-Yates & Gonnet's bit parallelism algorithm, which shifts the calculated state to the left by 1 bit (State[j-1]<<1). The search engine has a time complexity O(m).

In addition, if a calculated state equals 0 when only a part of the characters in the pattern have been loaded, it is not necessary to continue reading the rest of the characters, because it is impossible to find a match in the range of text. A quick state check can be performed after each state calculation.

Finally, the post-processor will read characters from 0 to (n-m) in the final state. It has a time complexity O(n-m+1). Also, if the search does not require knowledge of the position of the pattern in the text, the post-processor can be omitted. If the final state is 0, it means the pattern is not in the text; otherwise, there must be at least one pattern in the text.

In a multiple processor environment, the text can be divided into multiple sum-strings, T = {t1, t2,…tr}, if the string is longer than the length of the computer word. Because the current CPUs are mostly 32-bit or 64-bit, and only some GPUs have 128-bits or 256-bits, this condition is easily met by the text. After dividing, the first (r-1) substrings have a fixed length, which is the length of the computer word. The last substring contains the rest of the characters (n - (r-1)w).

The pre-processor has exactly the same process as the algorithm working on the single processor, and there is no data exchange between the different processors. In the search

engine, each processor will continue to process its own substring. However, some data is required from other processors. Therefore, the state calculation formula can be changed to the following formula.

State[c][j] = State[c][j-1] & (S[$c+q$][j] >> (j%w)) & (S[c+q+1][j] << (w-j%w));
if(j== (q+1) w) { q++; }
where q is a index integer and the initial $q = 0$;

All shift variables in the formula are from the result of the pre-processor, and not the calculated state from the search engine. It requires a calculation to get the shift size, not using the default 1. If it shifts the calculated states, it may reduce some waiting time to synchronize different processors. Because threads are controlled by the operation system and a single thread usually does not monopolize a single processor, it is hard to keep all processors synchronized all the time.

An example is given in Figure 49. In this example, n>m>w and contains almost all possible situations in a multiple processor environment. The text contains two copies of the pattern: one is at position 2 and another is at position 17. In the processor 1, the search engine will stop after the second character of the pattern, because State [1][2] = 0. The following calculated states require "AND" with State [1][2], which will not have bit "1" any more. In the processor 3, this processor is only responsible for characters T[24…27], but the maximum possible position for the pattern in the text is (n – m) = (28 – 10) = 18. This processor does not need to perform the search after the pre-processor.

In general, there are two ways to perform multiple patterns matching based on the Reversed Bit- Parallelism algorithm. The most basic method is the Brute Force. It will use the single pattern matching in the previous section to test all patterns one by one. The advantage of this method is that any pattern can be tested without any pre-process, so therefore any new pattern

can be directly tested. A second choice is converting all patterns into an automaton, and running over all nodes on the automaton. In this case, all the patterns require the pre-process, and the number of total nodes is smaller. If the automaton is not very big, all node information can be saved in the cache; if the automaton is too big, some internal information will need to be saved in the memory which may slow down the speed.

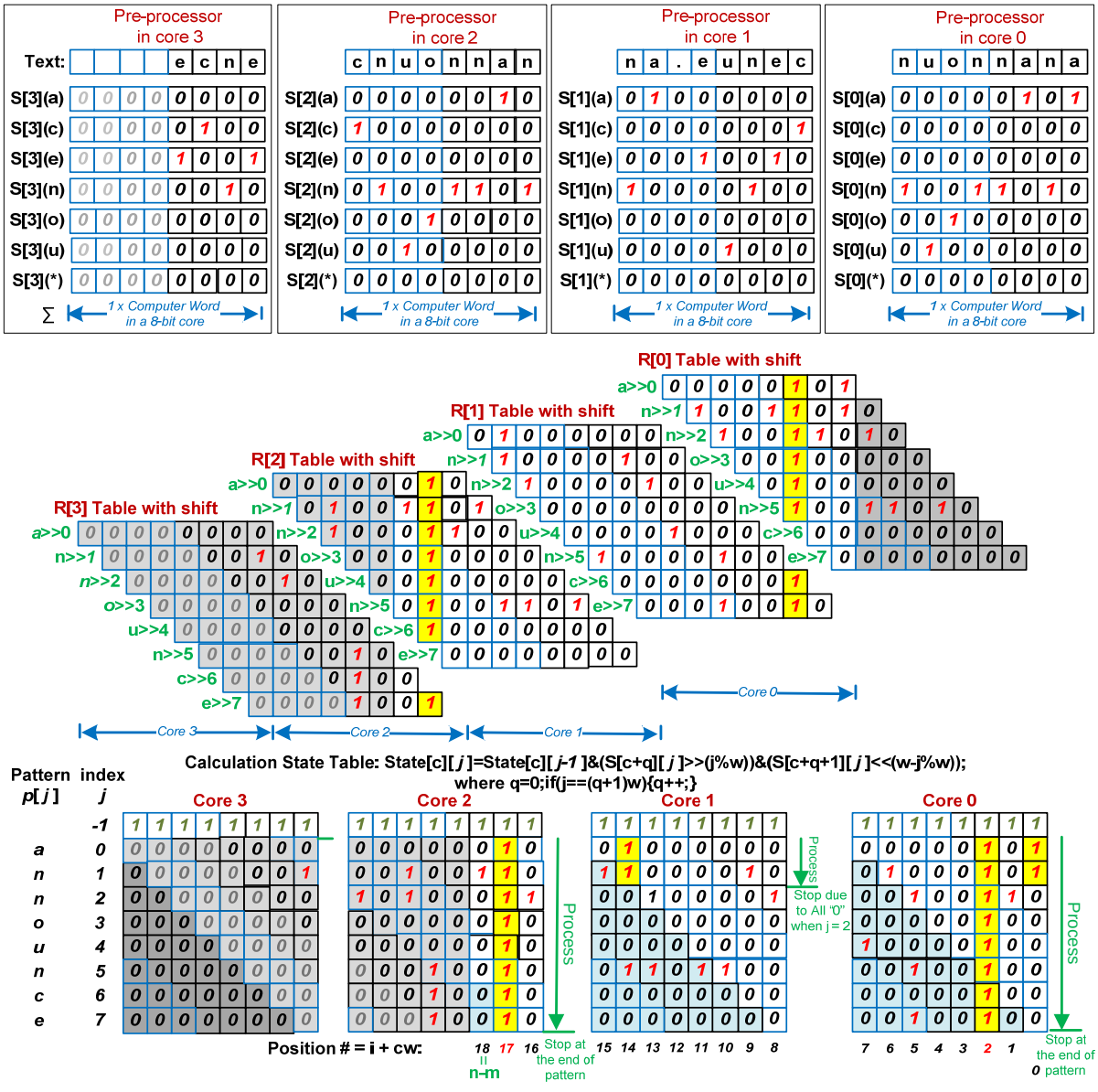


Figure 49: Searching pattern "announce" in the text "anannouncenue.annannouncence" with four-core in a 8-bit computer by the Reversed Bit-Parallelism (Reversed Shift-AND) Algorithm, n = 28, m = 10, c = 1, w = 16. Characters are encoded in ASCII.

RBP is an algorithm based on the Bit-Parallelism algorithm. It has inherited some characteristics from Bit-Parallelism. A comparison of them is shown in Table 3.

Table 3: RBP vs. Bit-Parallelism (BP)

Characteristics	RBP	BP
Time complexity of pre-processor	=$O(n+\sigma)$, (longer)	=$O(m+\sigma)$, (shorter)
Time complexity of search process	<=$O(m)$, (shorter)	=$O(n)$, (longer)
Post-processor	<=O(n-m+1), but steps are very simple and fast, and not required if no matching pattern in the sub-text.	Not required
Complexity of process in pre-processor	Very simple	Very simple
Complexity of process in search process	Very simple	Very simple
Complexity for complement symbols in the pattern	Easy	Easy
Complexity for a range of characters (e.g. 0-9, a-z) in the pattern	OK	Easy
The maximum number of threads/processors which can be used in the search	=n/w+1	Depending on the total length of pattern(s)

For many advanced pattern matching algorithms, such as automata algorithms, sliding window algorithms and bit parallelism based algorithms, their preparation process has been done in off-line in most applications, which does not cost the pattern search time (in-line processing time) for each individual pattern search. The pre-process is only required once for the search of multiple know patterns with different coming (unknown before read) text. However, RBP requires a quick pre-process on the text in such application. RBP is good at some applications such as text analysis, which searches multiple coming patterns with a known text.

5.3. Parallelized Aho-Corasick (PAC) Algorithm

As a multiple patterns matching algorithm, the Aho-Corasick (AC) algorithm is widely used in the intrusion detection area. It uses automata technology and spends a large amount of time in analysing the patterns and building a DFA in the preprocessor. Then it can have a very fast speed to perform the search in different texts regardless of the size of patterns. The time complexity of the search is only O(n) or O(n x log σ) if the automaton is stored in a direct

access table. Although the preprocessor requires a long time, it can be performed off-line in most applications.

The original AC algorithm uses a single character as a unit to build DFA, and then parses the text with calculated DFA based on the single character unit. As a single character is encoded in 8 bits, a 32 bits processor word can fill 4 characters in total and a 64 bits processor can process eight characters in total. Hence, the idea of paralleling the Aho-Corasick algorithm is to utilize multiple characters ("d" characters) into one character set. Then the single job can be divided into c equal parts. The maximum of "d" is "w/8", e.g. 4 in the 32 bits processor; and 8 in the 64 bits processor.

An example (d = 2) is shown in Figure 50. A text with 15 characters has a time complexity O(15) in a single core system with the original AC algorithm. With the parallelized AC algorithm in a dual-core system, both cores spend O(7) to finish the search process. If some character set can be filled up by the pattern characters, the empty bits can be filled with "0" to build the DFA, e.g. the last character "e" in the pattern in the example of Figure 50.

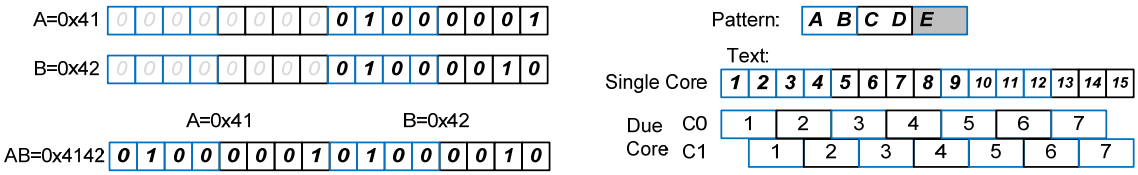


Figure 50: Parallelized Aho-Corasick (PAC) algorithm with 2 cores, d = 2.

However, a big "d" size will have a much bigger automaton and they have an exponential relationship. A relationship between the number of states in the automaton and the size of "d" is shown in Table 4. By default, the AC algorithm uses the states content as the index of all states; therefore, an empty state will still need a memory space. From the table, it is easy to see that only "d = 2" is acceptable at this point owing to the memory usage. A solution to the huge memory consumption is hashing.

Table 4: The number of states in the automaton vs the size of "d" in the PAC algorithm.

No. of characters in character set - "d"	Number of states in the automaton	Size of memory in 32-bit OS	Size of memory in 64-bit OS
1	$((\mathbf{2})^{\mathbf{8}})^{\mathbf{1}} = 256$	5 kB	7 kB
2	$((\mathbf{2})^{\mathbf{8}})^{\mathbf{2}} = 65{,}536$	1.25 MB	1.75 MB
4	$((\mathbf{2})^{\mathbf{8}})^{\mathbf{4}} = 4{,}294{,}967{,}296$	80 GB	112 GB
8	$((\mathbf{2})^{\mathbf{8}})^{\mathbf{8}} =$ 18,446,744,073,709,551,616	Huge	Huge
d	$((\mathbf{2})^{\mathbf{8}})^{\boldsymbol{d}}$		

In the Parallelized Hashed AC (PHAC) algorithm, the group of a character set can use a hash algorithm (e.g. CRC) to map into a small table, e.g. 64 bits can map into 16 bits or 8 bits. With hashing, there is a possibility that a mismatch will be detected as a match. Therefore, a full single pattern matching is required at the end of PHAC when a match has been detected.

5.4. Testing and Results

Some of the above pattern matching algorithms have been implemented, and tested on a 64-bit computer with Linux 2.6 x64. All programming has been carried out in "C" programming language.

All the following tests have been carried out on an Intel i7 computer with two CPU sockets. The CPU is Intel Xeon E5520, 4 cores with 2.26 GHz running frequency with hyper-thread enabled. The CPU has two 128 KB L1 caches; one 1MB L2 cache; and one 8MB L3 cache. The system has eight banks of 2GB DDR3 1067 MHz memory. The processor has a 5.86 GT/s memory accessing speed. The system has in total 16 threads and the Linux system BogoMips is 4533 for each thread.

Because all tests are running in a real computer environment, many factors may affect the results, such as memory usage, hard drive usage, network, system kernel programs and other user space programs. During the tests, all test processes have been set with a high priority to

reduce the influence from the user space programs. However, some kernel space programs and high priority system programs may also affect the test results. In addition, all results are an average value of three tests.

Furthermore, in the pattern matching tests, the test results may show a big difference in different patterns and tests. All test results in this chapter are approximate values. However, they can still provide a general idea about the performance of the different algorithms.

5.4.1. Single Pattern Matching in a Single-Core System

For the single pattern matching algorithms, three different algorithms, including Bit-Parallelism (BP), Reversed Bit-Parallelism (RBP) and Horspool, have been evaluated.

In the evaluation, the recorded time includes all in-line processes and no off-line processes. For example, the time of RBP includes both the pre-process and search process; the times of BP and HorsPool only include the search process and not the pre-process. At the end of the search process, all occurrences and the location in the text will be returned.

Test 1: Search for a single 7 characters pattern in variable length of texts, the alphabet $\sum$ is 256. There is only one occurrence in each search. The result is shown in Figure 51.

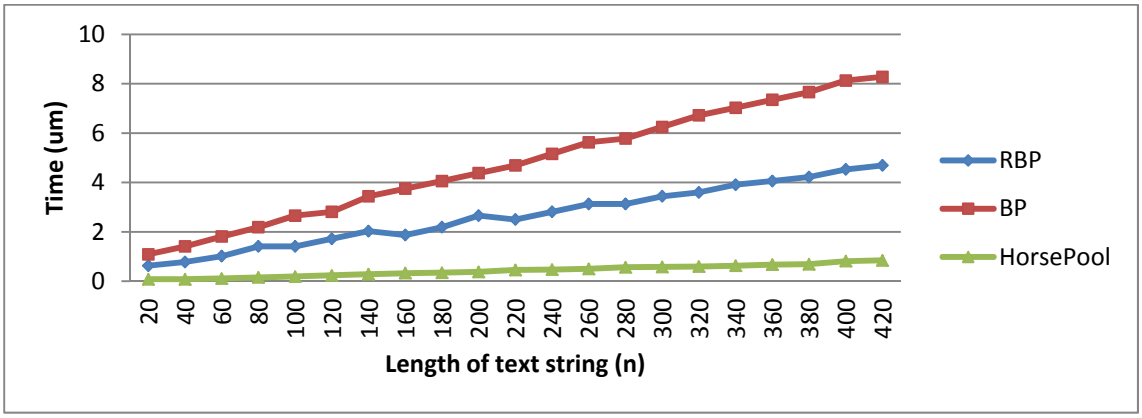


Figure 51: The performance comparison of single pattern matching algorithms when the length of pattern is 7 characters, and the length of text varies.

Figure 51 shows that HorsPool is still the fastest search algorithm in this test, RBP is slightly faster than BP.

Test 2: Search for a single 3 character pattern in variable lengths of texts, the alphabet ∑ is 256. There are multiple occurrences in each search. The result is shown in Figure 52.

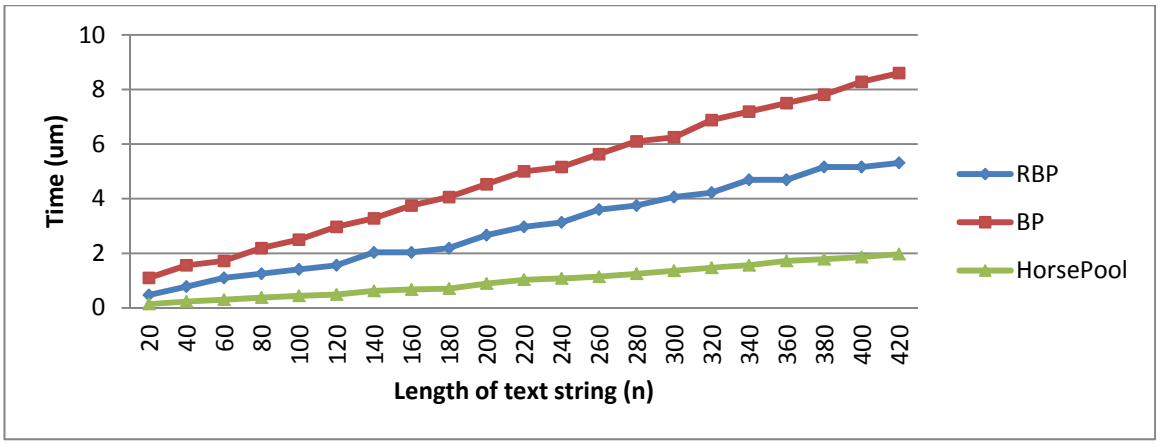


Figure 52: The performance comparison of single pattern matching algorithms when the length of the pattern is 3 characters, and the length of text varies.

Comparing Figure 51 with Figure 52 shows that HorsPool has a slower search speed with multiple occurrences in the text. Both BP and RBP have similar results to Test 1.

Test 3: Search for a single 64 character pattern in variable lengths of texts, the alphabet ∑ is 1. There are multiple occurrences in each search. The result is shown in Figure 53.

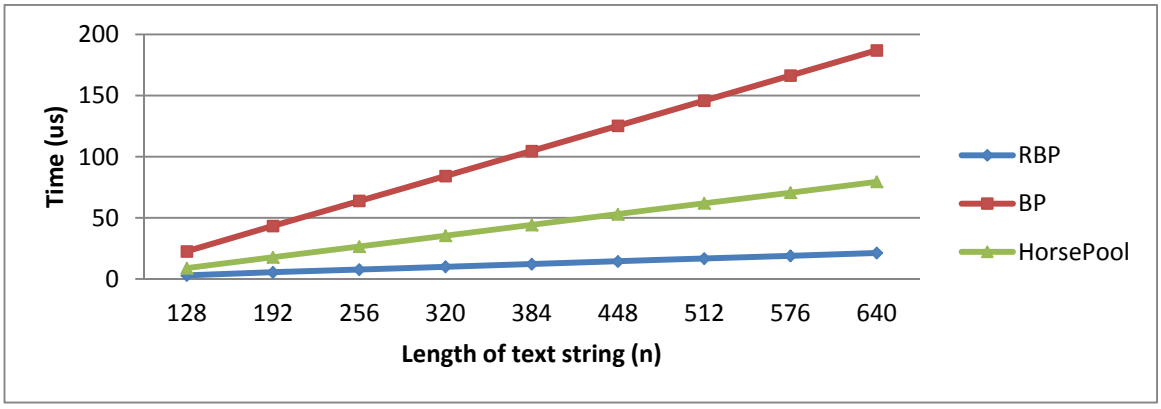


Figure 53: The performance comparison of single pattern matching algorithms when the length of the pattern is 64 characters, and the length of text varies. There are multiple occurrences.

With a decrease in the size of the alphabet, the number of occurrences in the matching increases, and the performance of HorsPool decreases. Test 3 gives the worst scenario of HorsPool. However, it is still better than BP, but worse than RBP. Therefore, HorsPool is a good single pattern matching algorithm when the alphabet is big, and the number of occurrences is small.

Test 4: Search for a single pattern with variable lengths in a fixed length of text with 640 characters. The alphabet ∑ is 256. There are multiple occurrences. The result is shown in Figure 54.

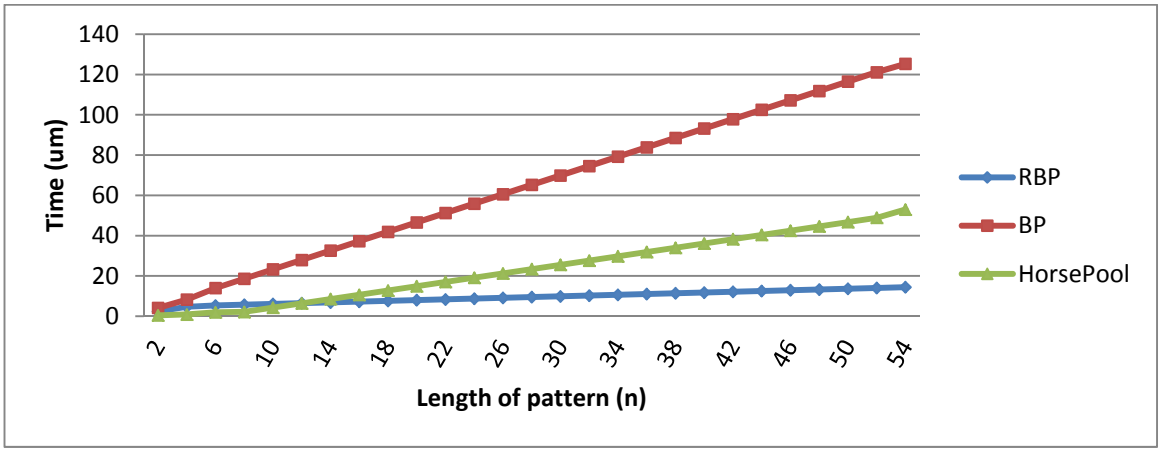


Figure 54: The performance comparison of single pattern matching algorithms when the length of text is 640 characters, and the length of pattern varies. There are multiple occurrences.

Figure 54 confirms that when the length of pattern increases, the performance of Horspool decreases faster than RBP.

5.4.2. Multiple Pattern Matching in a Single-Core System

In the test of multiple pattern matching algorithms, the following algorithms are included in the testing:

- BP for multiple patterns
- RBP with Brute Force
- RBP with DFA automaton

- Aho-Corasick (AC) for DFA, the “acsmx” module of Snort.

The time spent by the pattern matching algorithm excludes all off-line pre-processes. At that time, all data have already been read into the memory.

Test 5: Search for a pattern set with 10 patterns in variable lengths of texts. The alphabet ∑ is 256. All patterns are in the texts. The result is shown in Figure 55.

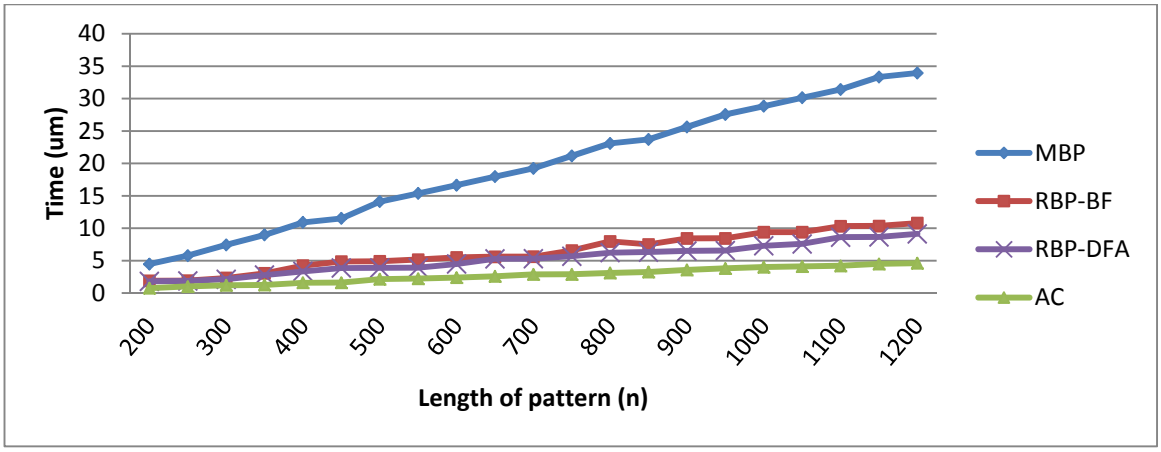


Figure 55: The performance comparison of multiple pattern matching. There are 10 different patterns, and all patterns are in the text.

Figure 55 shows that AC has the fastest matching speed, multiple patterns BP is the slowest algorithm, and the speeds for RBP with Brute Force and RBP with DFA are very similar.

Test 6: Search for a pattern set with 100 patterns in variable lengths of texts. The alphabet ∑ is 256. All patterns are in the texts. The result is shown in Figure 56.

Figure 56 shows that AC has the best speed, the time spent by multiple patterns BP increases greatly. The speed of RBP is also slower; however, RBP with DFA automaton has a better speed than RBP with Brute Force.

Therefore, all above tests shows that AC is the best pattern matching algorithm in a single process environment.

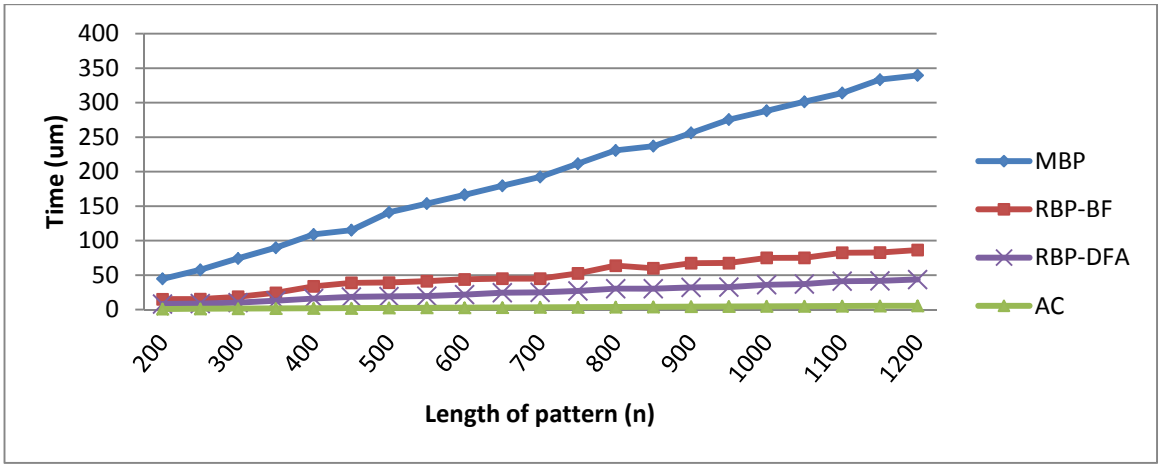


Figure 56: The performance comparison of multiple pattern matching. There are 100 different patterns, and all patterns are in the text.

5.4.3. Multiple Pattern Matching in a Multi-Core System

In the multi-core system, the programs use "fork()" to create multiple processes. "mmap" shared memory is used for exchanging big information; "signal" is used to send the one-bit message between different processors. In the shared memory, "semaphore" with "busy wait" is used to avoid data race and deadlock.

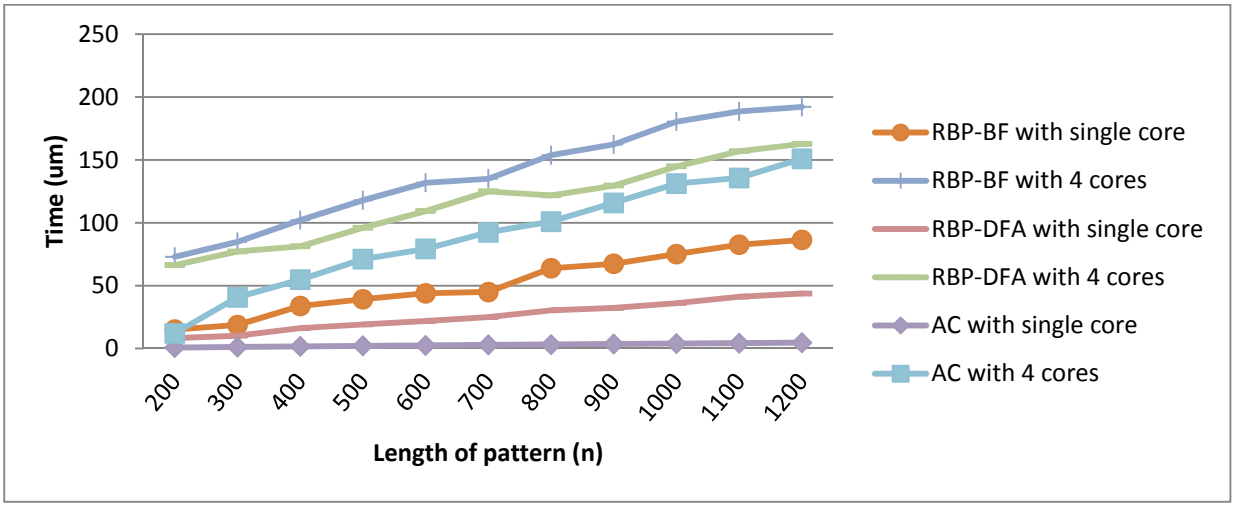


Figure 57: The performance comparison of multiple pattern matching. There are 100 different patterns, and all patterns are in the text.

Test 7: Search the same pattern set with Test 6 in variable lengths of texts by multiple processes on a multiple core machine, the alphabet ∑ is 256. All patterns are in the texts. The result compared with the single process result in Test 6 is shown in Figure 57.

Figure 57 shows that all parallelized algorithms have a slower matching speed than the original sequential algorithms owing to the small granularity of the text length. A large amount of time is spent on the memory operation. Therefore, the pattern matching algorithms are not suitable to be parallelized for network packet inspection on a general computer.

5.5. Conclusion

In the single pattern matching, HorsPool is still the best pattern marching algorithm when the number of matches is small.

In the general computer, RBP is slower than HorsPool most of the time owing to the pre-processor. RBP spends a large amount of time and memory for the pre-processor to read the text, and the total matching time will be large when the text is very long. In addition, RBP is not good at handling a large size alphabet. Some unicode encodings, such as the 16-bit, variable-width UTF-16 and 32-bit, fixed-width UTF-32, will require very big memory. By dividing a single unicode encoding into multiple parts and processing them in sequence, the size of memory can be reduced. For example, a 32-bit encoding can be divided into four 8-bit encodings. Then the total memory cost will be only 4 x 255 x w bit. As a Bit-Parallelism algorithm, RBP may be more suitable on a hardware processor, such as FPGA, which will have less time to organize the bits in the memory. And the tests show that RBP has a faster matching speed than the traditional BP algorithm.

A more detailed comparison between the RBP and a Windows-shift algorithm, such as Horspool, is shown in Table 5. Windows-shift based algorithms beat the process speed on the matching possibility. They can easily ignore those unnecessary matches, and safely shift the window. RBP can simulate a “windows-shift” by checking the “calculated state”, if the “calculated state” is 0 in the shift-AND algorithm, the following comparisons can be ignored. It may speed up the matching process when there is no matched pattern in the text.

Furthermore, when the length of a word becomes larger, e.g. from 32-bit to 64-bit, the performance of window-shift algorithms will not increase, but the overall performance of RBP will be increased. However, a serious disadvantage of RBP is that the preprocessor of RBP is in-line in most applications.

Table 5: RBP vs. Windows-Shift (e.g. BM, KMP, Horspool)

Characteristics	RBP	Window-Shift
Time Complexity of the Pre-processor	=$O(n+\sigma)$, (slower)	=$O(m+\sigma)$, (Horspool)(faster)
Time Complexity of the search process	<=$O(m)$	<<=$O(nxm)$, (Horspool) Depending on the possibility (the size of alphabet)
Post-processor	<=O(n-m+1), but steps are very simple and fast, and not required if no matching pattern in the sub-text.	Not required for most algorithms. However, =O(m) for some algorithms (e.g. Wu-Manber) to conform the matching
Longer computer word (w)	A longer w can help the algorithm to process more characters in one clock cycle. Can reduce the number of threads required in the search.	Does not help the algorithm to increase the speed
Safely ignores some unnecessary matches	Stops to match the remaining characters in the pattern when the calculated State (variable) is 0. Fixed length w.	Through Window shift. Varied length, depending on the shift algorithm.

Therefore, RBP may be unsuitable for some applications, which already have all defined patterns before the search process. It may be used in an environment in which the next pattern is chosen or generated after having received the search result of previous patterns. An example is the "Intrusion Pattern Discovery Module" designed by Lih-Chyau Wuu [44] which is mentioned in the background session. In such a system, when some new rules are generated, the system will take a very long time to add these new rules into the original rule set, and compare the new rule set to the RTN and OTN database. RBP may be used during this time.

In the multiple patterns matching algorithm, the Aho-Corasick (AC) algorithm is indisputably the best algorithm in a single core machine. The slowness of the RBP is due to the online pre-

processing of the text string. RBP with DFA is a little faster than RBP with Brute Force. However, the bigger memory consumption on the calculated states slows down the processing speed.

In the multi-core environment, none of the parallelized algorithms mentioned in the previous session wins the test. Most existing parallelization APIs, such as Pthread, OpenMP, and shared memory have a big overhead. With a big granularity, the overhead can be ignored; however it is a big issue for small granularity of data. In such applications, most of the time is spent on the memory operation rather than calculation. On a hardware implement, such as FPGA, the above parallelized algorithms may be applicable. However, some further tests are required on the feature.

In the application of intrusion detection, the traffic has millions of packets per second in the gigabit Ethernet network. The granularity of each packet is extremely small compared with most other parallelizable applications. Therefore, the sequential AC algorithm is still the best option for deep packet inspection, such as the IDS, and IPS system on the general computer.

6 Pipeline and process management system

With a small granularity, parallelized processing will require more time than sequence processing. By increasing the granularity, the overheads can be reduced. In the intrusion detection program, data is processed at packet level. To increase the granularity, a section/function parallelization of Snort will be discussed in this chapter.

6.1 Introduction

The goal of this project is to force more and more processors and cores to work together in the whole packet processing process in the application of Snort.

Some big programs, such as Snort, have a very complex structure. Some parts of the process can be easily parallelised and some parts have a dependent relationship with a previous result. Therefore, a pipeline system can be used to segment the whole process. However, a good pipeline system requires a similar load on each of the segments to reduce the waiting time for the previous results. As described in the discussion in Chapter 3.1, the work load on different parts of Snort is not evenly distributed, and changes dynamically depending on the traffic.

A group managed pipeline structure cannot only segment the whole process into different pipelines, but also parallel the work load on each single segment by different parallelization methods to speed up the heavy load segment.

Amdahl's law indicates that the maximum speed up of a parallelized program depends on the percentage of the unparalleled parts. The job of the group managed pipeline structure is to reduce the size of any unparalleled part and try to use different methods to parallelize different segments of a program, rather than one method to parallelize the whole program.

6.2 Pipeline and Buffer

Pipeline is a set of series of processes connected by a group of data processing elements. It is a parallel processing method which is first used as an assembly line to increase the overall throughput. It requires a buffer between two continuing segments, and when an object goes through the pipeline, some extra time is required in the buffer. Therefore, the overall processing time for any particular packet is longer than a sequential process.

As discussed in the previous chapter, Snort consists of a few different parts: packet capture (PCAP), decoding, packet preprocessing, packet detection, post-processing, and reassembled packets reprocessing. The left flowchart of Figure 58 shows the sequential processing of the officially released Snort 2.8.

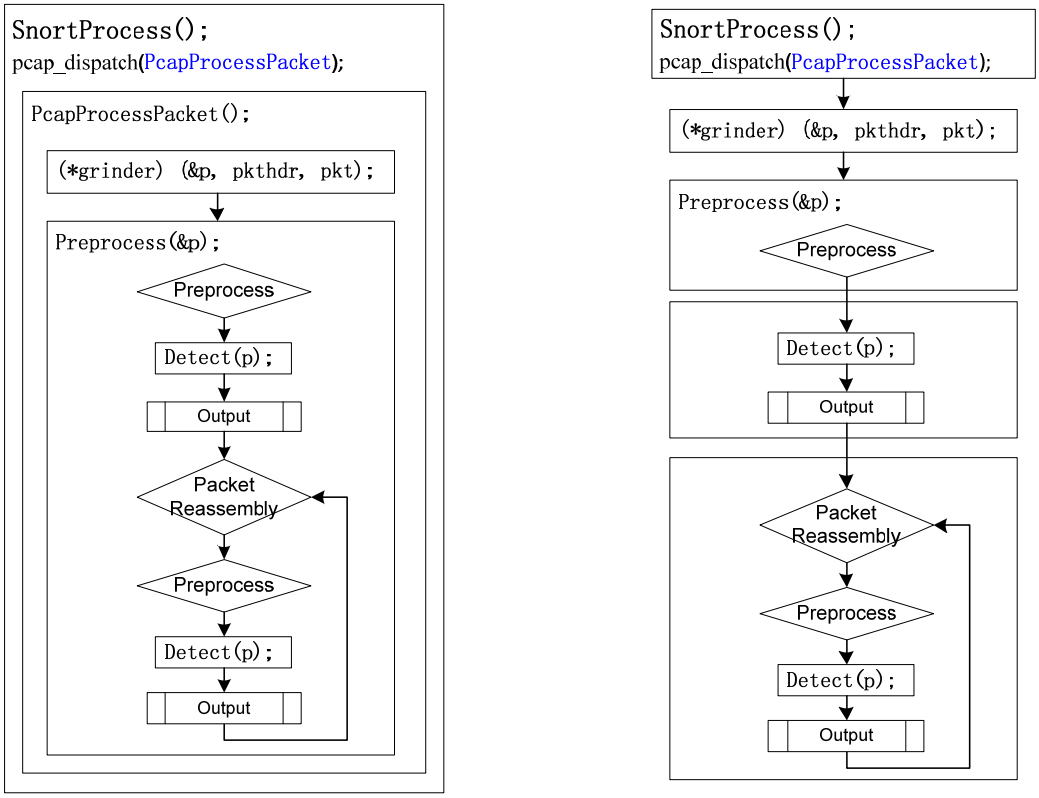


Figure 58: The processing flowchart of Snort, the left flowchart shows the structure of official Snort 2.8; the right flowchart shows the pipeline structure of Snort.

PCAP reads packets from the network card, copies the reading packet into a buffer and calls the callback function of Snort directly to process the captured packet. Then, the packet decoder will convert packet data into a readable format. There is no dependency relationship between any contiguous packets in the decoding process. Packet pre-processors inspect the IP header part and a small amount of the data part of packets. Then packets changed by the pre-processors will be passed to the next detection stage, except for some special packets which will stop here.

The detection engine normally performs some different types of multiple patterns matching, marks any triggered signatures and also performs the no-content pattern matching by the detection plugins. Some fragment packets will be bypassed by the detection engine. And then the post-processors will output the events in the configured methods, such as output files, syslog server and database server.

Finally, all fragment packets will be reassembled and sent back to the preprocessor. This operation has a dependency relationship with some contiguous packets. It needs to access a previous packet buffer and all packets coming into this stage require the result from the previous preprocessing and detection, and some packets will be modified by those sections. Therefore, this part cannot be processed in parallel with the previous detection section.

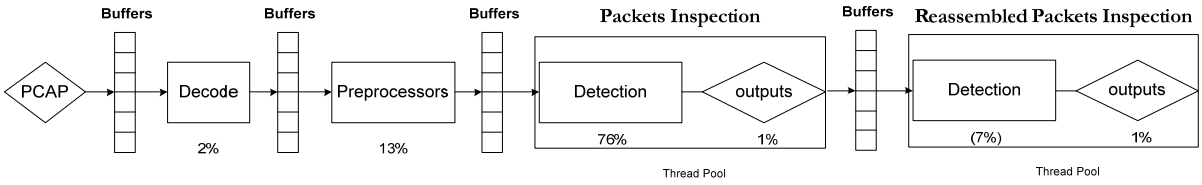


Figure 59: The pipeline process of Snort.

Normally, the whole process can be divided into five parts: PCAP, packet decoder, preprocessors, detection & output, and the reassembled packets process as a stand-alone section. A process structure is shown on the right side flowchart of Figure 58 and Figure 59.

The buffers in the middle of two consecutive sections are stored in the shared memory. A circular buffer can be used to reduce the lock and mutex of multiple processes accessing the shared memory, and also can avoid the data race and deadlock. An example of the circular buffer is shown in Figure 60. Each circular buffer has one producer and one consumer and each of them works independently, and points to the current working position in the buffer. Only a producer can "write" a new packet into the buffer. Generally, there are three different buffer states: empty, full, and neither empty nor full. The conditions of the buffer states are listed in Table 6.

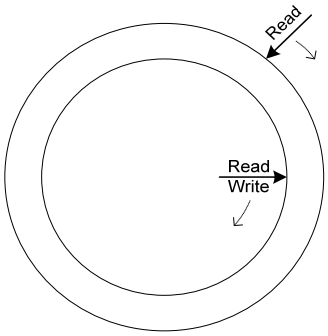


Figure 60: The circular buffer of the pipeline.

Table 6: The states of the circular buffer.

Buffer Condition	**Buffer States**	**Note**
Producer_index == consumer_index	Empty	Writeable but not Readable
((producer_index + 1) % BUFFER_SIZE) == consumer_index	Full	Readable but not Writeable
Others	Neither empty nor full	Readable and Writeable

There are also two types of circular buffer: "data buffer" and "address buffer". When LibPCAP read the new packets from the driver, the whole packet data and the header information need to be saved to a shared memory buffer for later inspection by other processes. The packet data should be kept in the buffer until it has gone through all pipelines,

and the PCAP should continue to capture new packets. The whole packet information for a single packet is kept in a struct “ePacketPack”, and the structure is shown in Figure 61. The “data” circular buffer is used to save this packet information, including captured packet data, its captured header, and decode packet information. A struct of “data” circular buffer “CircularDataBuffer” is shown in Figure 62. After all inspection processes have been carried out, the saved packet will be deleted at the end of the pipeline. The circular buffer is located in the shared memory, and can be accessed by any of the Snort processes.

```
struct pcap_pkthdr {          /* original captured packet. Defined in pcap.h */
 struct timeval ts;           /* time stamp */
 bpf_u_int32 caplen;          /* length of portion present */
 bpf_u_int32 len;             /* length this packet (off wire) */
};

typedef struct _ePacketPack{      /* a buffer keeps all packet information  */
 void* cdb;                       /* point to the buffer address */
 struct pcap_pkthdr pkthdr;       /* captured packet header */
 u_char pkt[PCAP_MAXETHPACKET];   /* captured packet data */
 Packet p;                        /* decoded packet information, some will point to  “pkt”
*/
 char* user;                   /*  user defined information when process packet */
 FragTracker ft;               /*  fragment parameter by  “frag2”  plugin */
 volatile long  usage;         /*  a record of space usage of data circular buffer */
 volatile long  status;        /*  a status information of data circular buffer */
} ePacketPack;
```

Figure 61: The struct of “ePacketPack”.

A PCAP header records the capture time of the packet in microseconds; and the packet size. Theoretically, the maximum size of the packet is 65536 bytes. However, a packet does not exceed its MTU during the transmission. If a bigger packet is sent by some device, it will be divided into a few fragment packets. And when Snort receives those packets, its “frag2” plugin will reassemble all fragment packets into one packet. This reassembled packet will have a maximum size of 65536 bytes. Therefore, the packet data buffer “pkt” should have a size of 65536 bytes. The packet buffer size required by each single packet is about 68 kB. If the circular buffer reserves space for 10,000 packets, the total memory required by the buffer is about 700 MB.

```
typedef struct _CircularDataBuffer {     /* data circular buffer for packets  */
 ePacketPack data[SIZE_OF_DATA_BUFFER];  /* a buffer keeps all packet information  */
 ePacketPack *pd;                        // producer location
 ePacketPack *cd;                        // consumer location
 volatile long  pi;                      // producer index
 volatile long  ci;                      // consumer index
 volatile long  di;                      // delete index
} CircularDataBuffer;

typedef struct _CircularAddressBuffer {  /* address circular buffer for packets  */
 void* adata[SIZE_OF_CIRCULAR_BUFFER];   /* a pointer of current processing packet  */
 void** pd;                              //producer location
 void** cd;                              //consumer location
 volatile long pi;                       //producer index
 volatile long ci;                       //consumer index
} CircularAddressBuffer
```

Figure 62: The struct of a circular buffer.

In the middle of the pipeline, when a pipe passes a packet to the next pipe, it is not necessary to pass all the packet information. Instead, a pointer pointing to "Packet p" can be used to pass the inspection job. This job list can be stored in the "address" circular buffer. The data stored in this circular buffer does not require deletion after reading; the producer can overwrite the previous data automatically owing to the data accessing prevention mechanism by the buffer states (i.e. empty, full, or other). In the pipeline, each processing unit in each pipe except the first pipe will hold its own address circular buffer as its job list, and it is the consumer of this list. The previous pipe is the producer of the circular buffer.

The control of the pipeline and multiple processes are created in the "SnortProcess" function in "snort.c" after all Snort initialization and loading all signatures. When a new process has been created, the child process will inherit all initialized variables from its parent process. After the initialization of the processes, the mother process will come into the packet capture loop. Its child processes will begin to perform their own job until a terminal message is received from their mother process.

6.3 Group Management Pipeline

Normally, each pipe of the pipeline requires having a similar job load to achieve the best overall performance. Otherwise, the pipeline with the heaviest load will become the bottleneck of the whole pipeline; the other pipes will have to wait for an available space on the buffer or a candidate packet in the buffer. However, it is difficult to divide the whole Snort into different sessions on average. The packet inspection session has a much larger work load than the other sessions. Therefore, multiple processes can work on these big work load pipes to share the load. An example structure is shown in Figure 63.

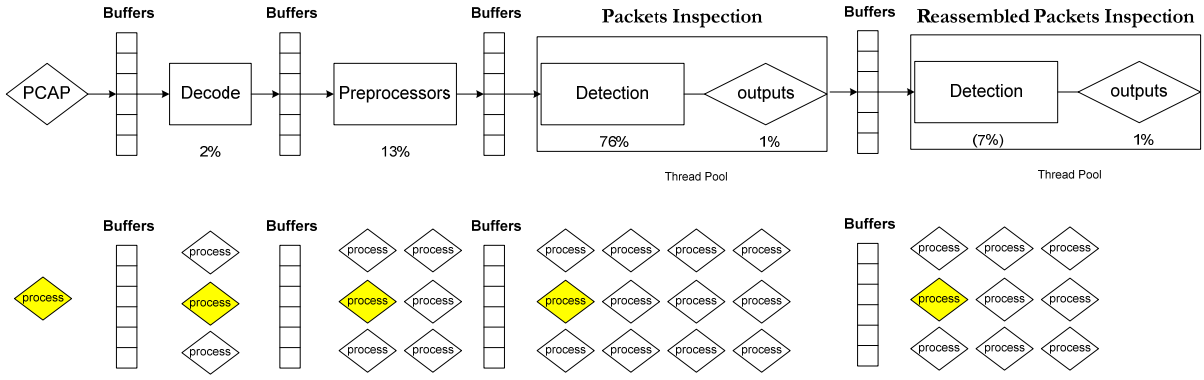


Figure 63: The pipeline process of Snort with multiple processes in some pipes.

In such a system, when a job is passed from multiple processes in the previous pipe to multiple processes in the next pipe, the pipeline becomes complex. The circular buffer only supports one producer and one consumer. For multiple producers or multiple consumers, it requires “mutex” or “lock” to avoid data race, which have much more overhead than the circular buffer.

By increasing the number of circular buffers, the multiple producers or multiple consumers issue can be avoided. In this pipeline, all processes have their own circular buffer instead of the main pipeline buffer. The structure of the pipeline is like a tree, which has only one trunk, more branches on the trunk, more sub-branches on each branch and so on. A structure is shown in Figure 64.

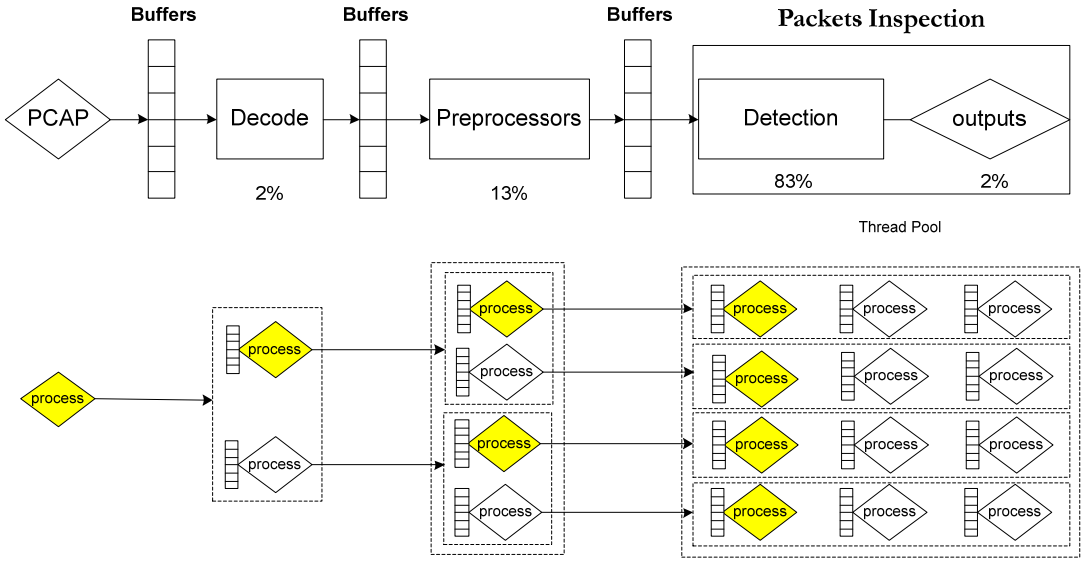


Figure 64: The pipeline process of Snort with multiple processes; all inspection and outputs are in one pipe.

The whole system is managed by multiple groups, and some groups also have some sub-groups as their memory. The innermost group of each pipe is built by a group of processes. All groups have their own group policy, which records some group features such as group manager, group members, reading buffers and writing destinations. All group members will work according to their group policy. Each group has a group manager. It must be a process, which has an ability to create or terminate a group member in its group. This feature can make the whole system easy and flexible for process management when the number of processes is large. A change in a single group does not affect the whole system too much.

In this system, a process can write into the buffer of all group memory from a group. The group members can only read their own input buffer. Therefore, any buffer will have one producer and one consumer and a circular buffer can be used as the buffer. An example is shown in Figure 65. Process A1 can produce new jobs for Processes B1 and B2 by writing the jobs into the input circular buffer of Processes B1 and B2. Process B1 can only read and process the job from its own input buffer. When Process A1 distributes jobs to different buffers, it will use a packet classification module to decide which buffer it sends the packets to, and the classification rule is recorded in the group policy of the destination group.

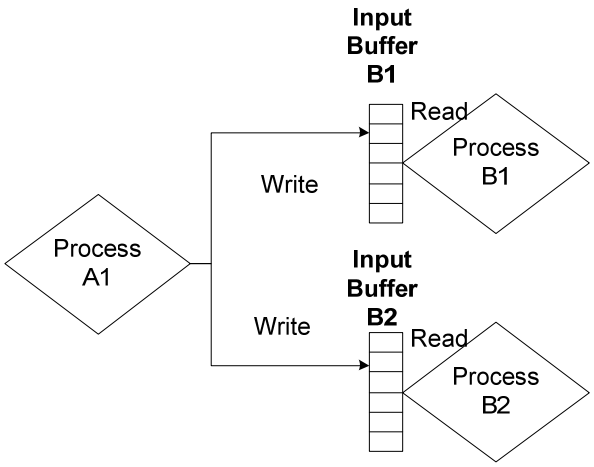


Figure 65: One processor writing into multiple buffers.

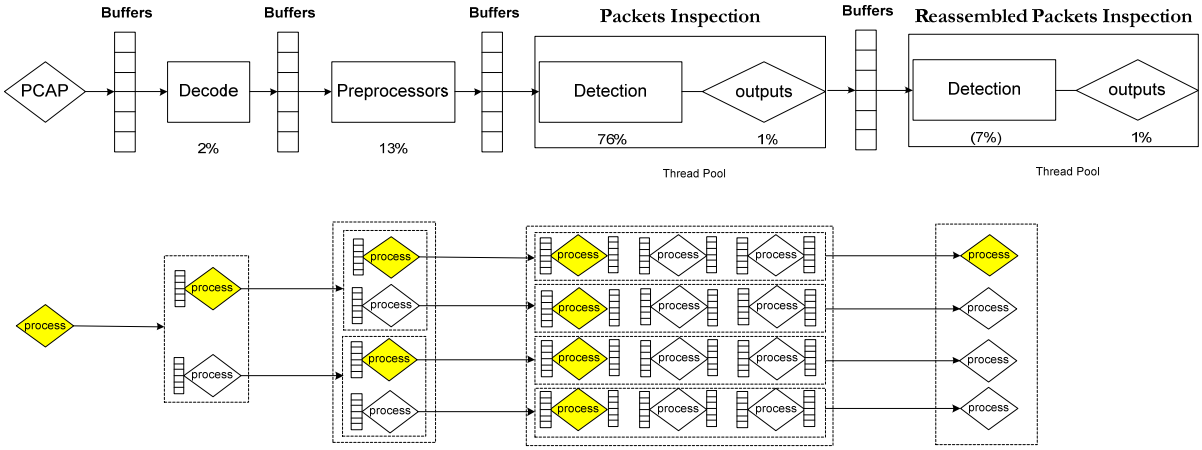


Figure 66: The pipeline process of Snort with multiple processes. Normal inspection and reassembled inspection are in two different pipes.

However, this system has a short pipeline and a big work load on the last pipe. According to the flow structure of Figure 63, the pipeline structure can be modified as Figure 66. In the new system, the final pipe has less workload than the previous pipe. Therefore, fewer processes are assigned to this job. If multiple processes push the job into one buffer, the buffer will have multiple producers. In this case, some processes have two buffers, besides the input buffer. They also have an output buffer. The new jobs are not pushed into the input buffer in the next pipe, instead, the new jobs are saved in the output buffer of the current processes, and the process from the next pipe will collect the buffer from the previous pipe. A buffer working example is shown in Figure 67. In the example, after Processes B1 and B2 finish their job, and want to shift the job to next pipe, they will put it in their output

buffer. And Process C1 will continue to check and collect the jobs in the output buffers of B1 and B2. Then the circular buffer can be used in this system.

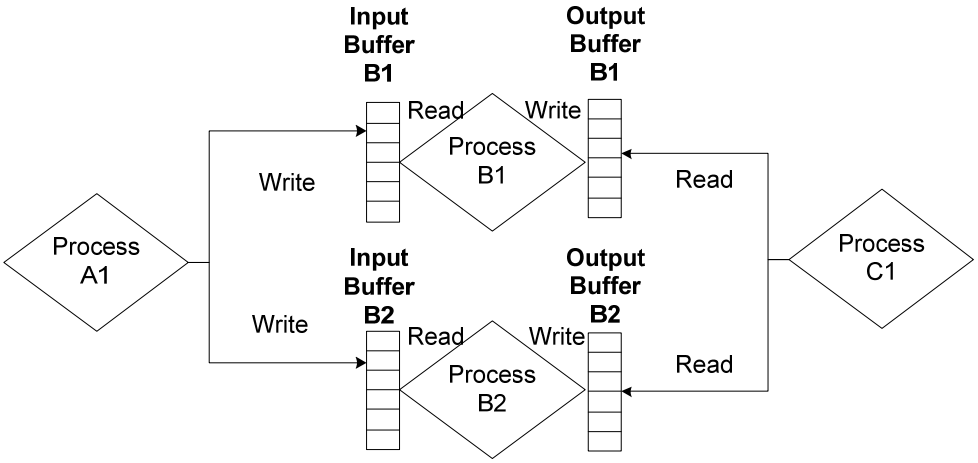


Figure 67: One processor collects (reads) data from multiple buffers.

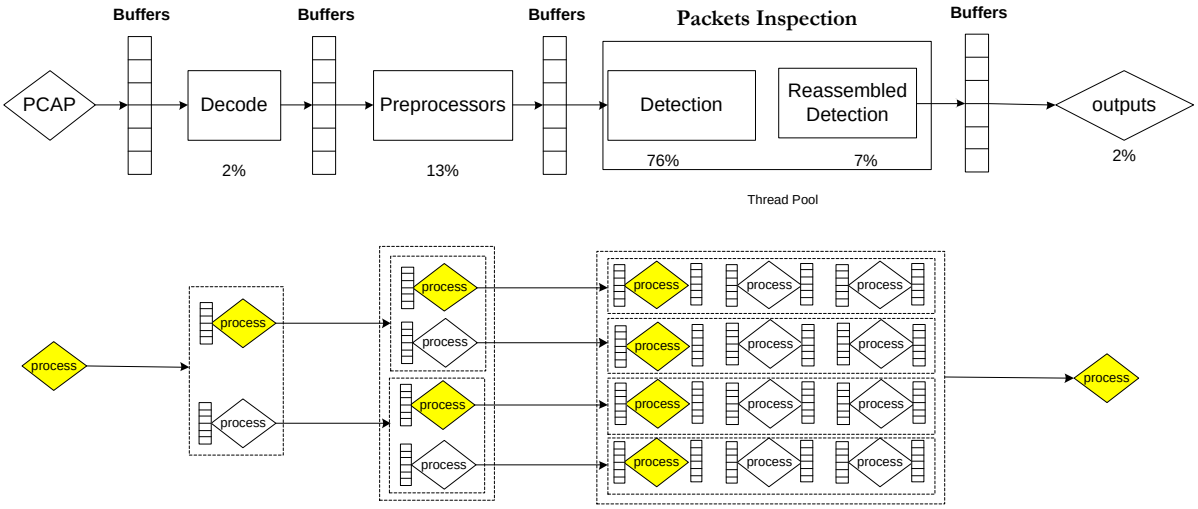


Figure 68: The pipeline process of Snort with multiple processes. All inspections are in one pipe and the outputs are in a different pipe.

In the network environment, if Snort outputs the alert to a remote server, such as an SQL server and the server stops working or loses response, Snort will also slow down or hang up because its output plugin will continue to search the remote server and try to send out the alert. Therefore, another option of the pipeline is separating the output module from the inspection process. A structure is shown in Figure 68.

6.4 Measurement on the parallel system

On the sequence program, a new incoming packet will be read and processed when the processing on the previous packet has been finished. It is easy to measure the processing performance by counting the number of coming packets (n_{packet}) and the time required by Snort to process these packets (t). The throughput (Tp) can be calculated by Equation 2.

$$Tp = \frac{n_{packet}}{t}$$

Equation 2

And an average processing time (T_{ave}) for a single packet can be calculated by Equation 3.

$$T_{ave} = \frac{t}{n}$$

Equation 3

In a parallel processing system, the measurement of instant performance is complicated for the following reasons:

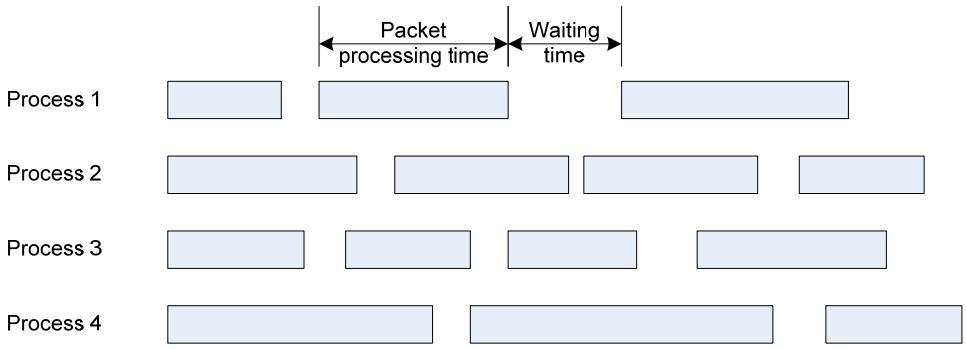


Figure 69: The processing time and waiting time in a simple parallelized system.

(1) The processing time for different packets is different. Even for a totally parallelized system, it is hard to count accurately the number of packets that have been processed.

Most of the time, part of a packet has been processed, but not a whole packet. Figure 69 shows a processing example on a simple parallelized system.

(2) The waiting time is different for different packets and different processes. Furthermore, the waiting time can be affected for different reasons, such as memory loading delay, PCAP capture delay, and the no-job delay (There are no jobs for a process, which can be caused by no traffic coming into the network adapters in a run time inspection, and the previous section not having done its job in a pipeline system).

(3) In a buffered pipeline system, each buffer may contain some packets, which have been done by some sections of the pipeline, but not by the whole process. An example of the buffer pipeline system is shown in Figure 70.

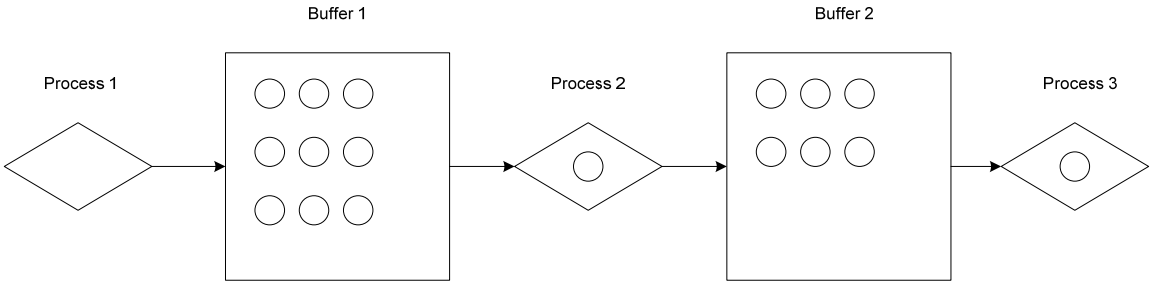


Figure 70: An example of a buffer pipeline system.

(4) Who is going to count and collect the counter information? In a parallel processing system, the message transmission and synchronization require extra time. Owing to the complexity of the counter calculation, the calculation also needs some CPU load if a periodic statistic is required. And the output of the statistic requires more time.

Compared with the sequence program, a parallel processing will have some different calculation formulas. Firstly, for a single packet, the total throughput does not equal the reciprocal of average packet processing time. For example, both throughput and packet processing time are larger than the sequence program.

$$T_p \neq \frac{1}{T_{ave}}$$

Equation 4

Secondly, the total throughput does not directly equal the throughput on a single process multiplied by the total number of processes.

$$Tp_{total} \neq Tp_{single\ process} \times n_{process}$$

Equation 5

To simplify the issue, a measurement with an acceptable accuracy can be carried out in the following way:

(1) In the pipeline system, the packet counting will be carried out on each individual section. Only count the incoming and outgoing packets and use the timestamp in the PCAP header to calculate the throughput. The counter variable can be stored in their local shared memory or a global shared memory. Equation 2 can be used by each individual process for its own calculation. The total throughput can be assumed to be the sum of each parallel limb.

(2) Choose a suitable sample time, e.g. 1 second, to increase counting accuracy.

(3) Use a separate process to organise the working processes, the calculation, and the output of the packet statistic periodically.

(4) Record the CPU usage for each individual core with the packet statistic.

(5) Record the buffer usage periodically.

In the periodic output, the packet processing has to be read with the related CPU core usage. When the CPU is not 100% used by the working core, it means that it spends some time waiting for a new job, and the previous sections require more computing power. The PCAP speed only indicates the PCAP capture speed, if the CPU core running PCAP is not 100% used. It means that the current traffic does not reach the maximum PCAP traffic allowed by the current hardware. This may be caused by either the hardware speed of network adapters or software traffic provided by the clients and services programs.

Generally, the packet throughput measured by each individual process indicates the average throughput in the sample period.

An overall inspection speed is decided by the slowest pipeline section, and also has a similar result to the PCAP speed.

6.5 Test of the Pipelined Snort

An Intel i7 system has been used for the tests. The computer has two CPU sockets and the CPU is Intel Xeon E5520, 4 cores with 2.26 GHz running frequency with hyper-thread enabled. The CPU has two 128 KB L1 caches; one 1MB L2 cache; and one 8MB L3 cache. The system has eight banks of 2GB DDR3 1067 MHz memory. The processor has a 5.86 GT/s memory accessing speed. The system has in total 16 threads and the Linux system BogoMips is 4533 for each thread.

The computer has a Linux 2.6.31-20-generic x64 operation system. All programming has been carried out in “C” programming language.

In the tests, multiple processes are created by “fork()”, and bind with different physical processors by CPU affinity. The test processes are set with a higher priority to reduce the effect from other programs.

```
+-------------------[Rule Port Counts]-------------------
|              tcp     udp    icmp      ip
|     src      801       9       0       0
|     dst     5688     390       0       0
|     any      180     115      39      11
|      nc       16       7      14       8
|     s+d        3       3       0       0
+-----------------------------------------------------
[ Port Based Pattern Matching Memory ]
+-[AC-BNFA Search Info Summary]-------------------------
| Instances        : 245
| Patterns         : 44861
| Pattern Chars    : 425552
| Num States       : 138663
| Num Match States : 17236
| Memory           :   6.76Mbytes
|   Patterns       :   2.12M
|   Match Lists    :   2.75M
|   Transitions    :   1.80M
```

Figure 71: Snort rules set used for the tests.

All tests use the default pattern matching algorithm "AC-BNFA". In the tests, Snort uses a rule set "snortrules-snapshot-2.8.tar.gz" which is a free Sourcefire VRT certified rule set released in July 2008 by Snort. There are 7226 rules (OTN) in this rule set linked into 259 chain headers (RTN). Detailed information about this rule set is shown in Figure 71.The computer has an Intel 82598EB 10 GbE card, and the card is connected by a PCIE v2.0 interface. However, it is hard to generate a 10 Gb traffic by a software as there is a very high load on processors when the traffic is bigger than a few hundred Mbps. Therefore, a pre-captured pcap file is used for the tests. The pcap file is captured by "Tshark" and saved in a memory drive for fast reading. The captured packets include some general network activities, such as web browsing, emails, FTP, ntp, p2p, file copy in the local network and so on.

The performance of pattern inspection may be influenced by many different factors, such as the CPU usage of other programs, rule set, test packets, or the memory usage. All results in this chapter are an average value of a few tests. Although it is an approximate value, it can still give a performance level of different implementation.

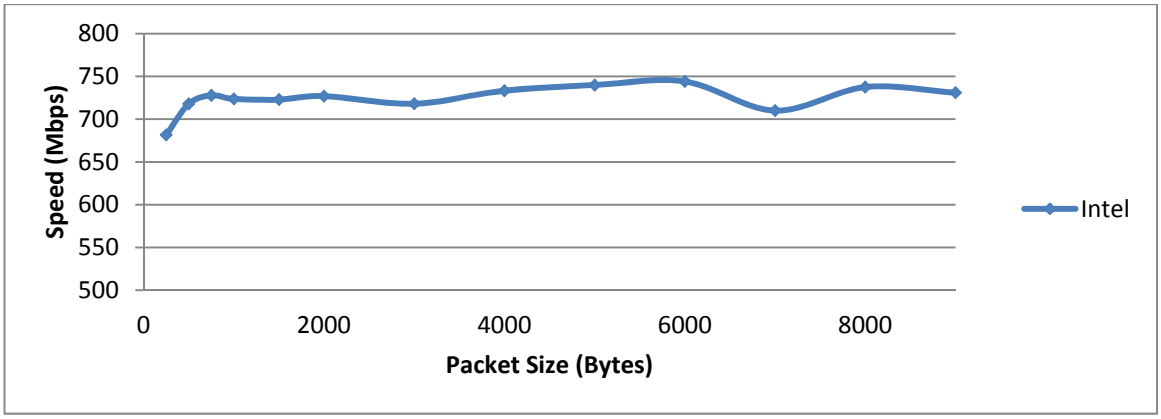


Figure 72: The inspection speed of original sequential Snort on the Intel computer.

All the following tests will use the original sequential Snort v2.8.5 as a reference and all modified versions of Snort are based on version 2.8.5. All test results from different versions of Snort are collected in the same environment and with the same data and rule set. A performance chart for the baseline sequential Snort on the Intel computer is shown in Figure 72. The test result shows that the original official sequential Snort has almost the same inspection speed for different sized packets.

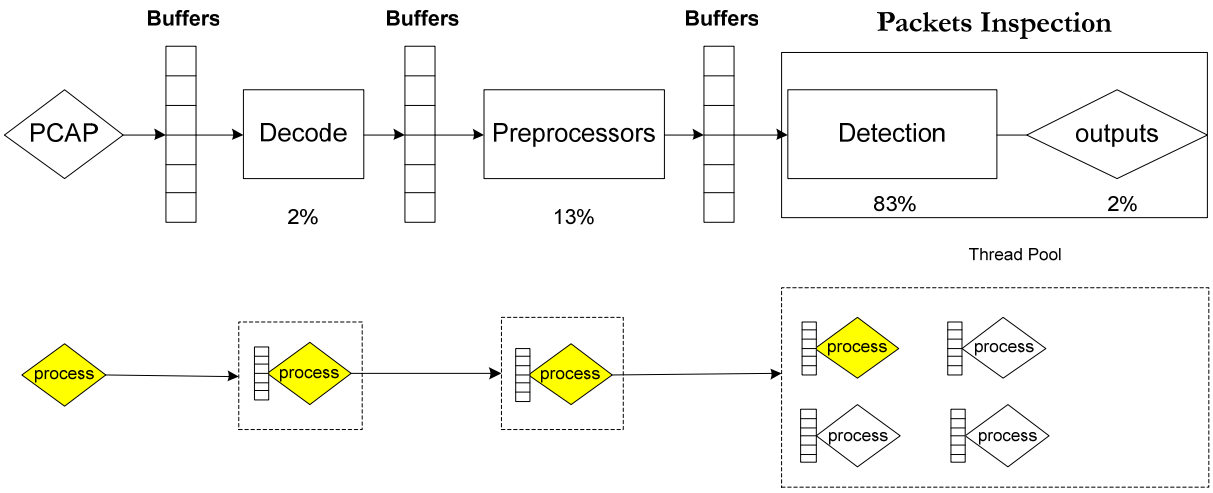


Figure 73: Snort test structure S01 (1+1+1+4).

With a pipeline implement shown in Figure 73, there are four pipes in the pipeline, only one process in the PCAP, decoder, and pre-processor, and four processes in the packet inspection module. An overall performance chart comparing it with the original official Snort is shown in Figure 74.

Figure 74 shows that the performance increases exponentially when the size of the packet increases. The inspection speed is very low when the packet has a length of less than 3000 bytes. It may be caused by a small granularity of the packets. With the small granularity, most time is spent on the packet copy, overhead data transfer, and memory operation initialization. When the size of the packet is bigger than 3000 bytes, it only has a slightly higher inspection speed than the original Snort. It looks as if the system reaches a bottle neck. In the testing, a 32768 packets data buffer and a 2048 packets address buffer are allocated. An average of a 20000 packets data buffer is used and the address buffer is almost 0 most times, which means when a new packet comes into the pipeline, it will be processed immediately. The bottleneck is not at the decoder, pre-processor, or packet inspection session.

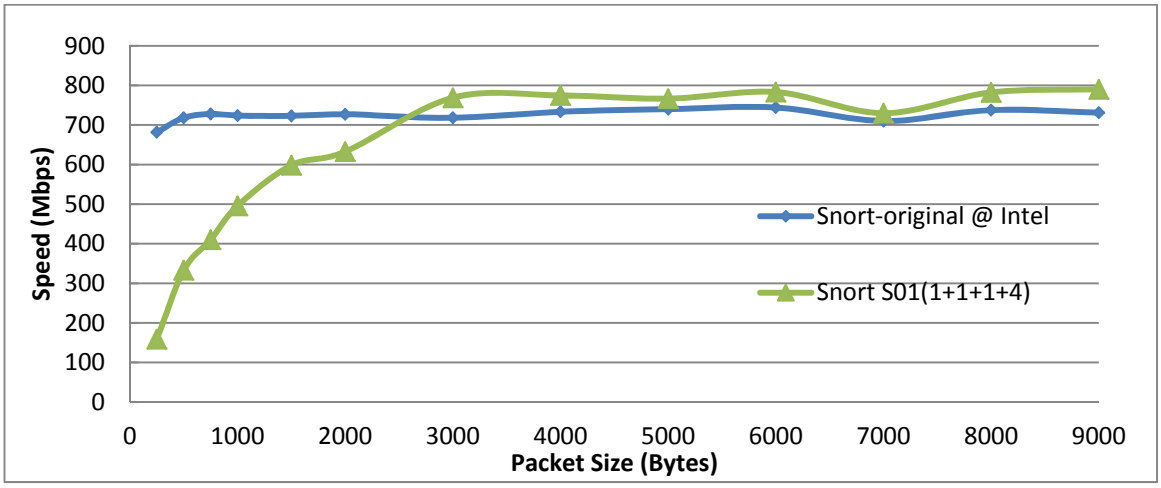


Figure 74: The inspection speed of Snort S01 (1+1+1+4).

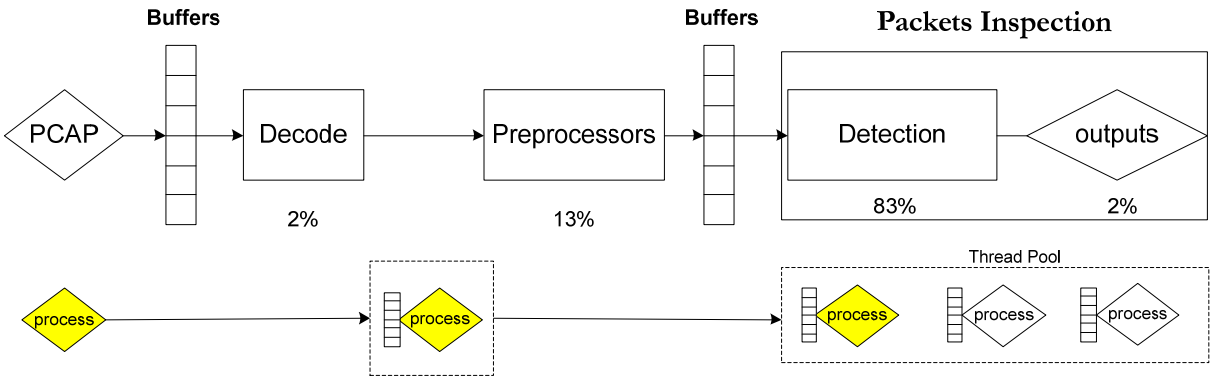


Figure 75: Snort test structure S02 (1+1+3).

A reduced stage of the pipeline showing in Figure 75 combines the decoder and pro-processor sections together and there is an overall packet inspection performance showing in Figure 76. It has very similar performance to the previous pipeline showing in Figure 73. It looks as if the system has a memory bandwidth limitation.

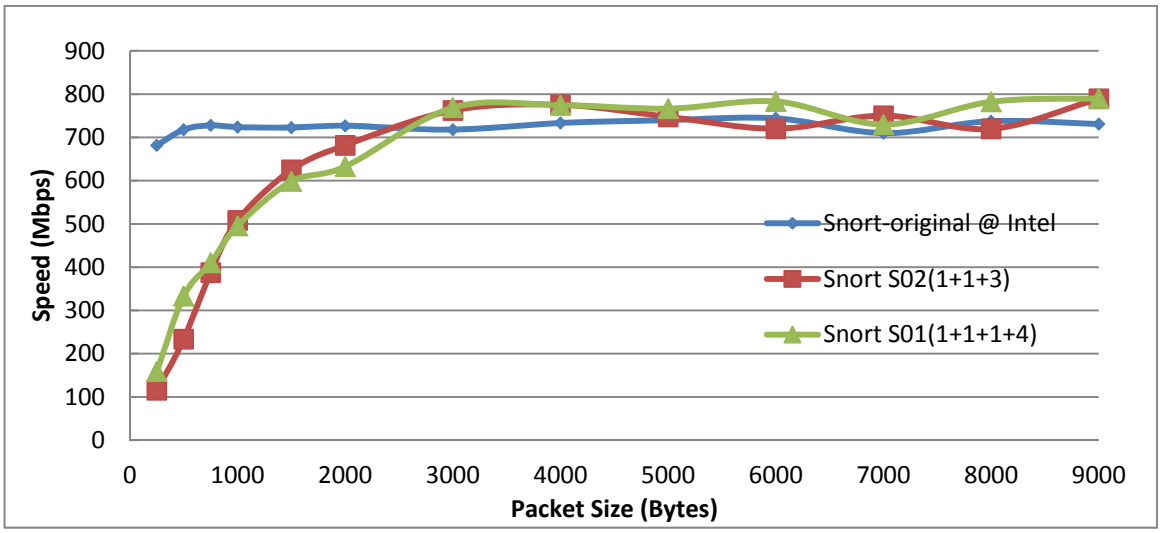


Figure 76: The inspection speed of Snort S02 (1+1+3)

Figure 77 shows the number of processes reduced to 2. A comparison performance chart is shown in Figure78. The figure shows that it has a better performance when the packet size is between 3000 bytes and 6000 bytes. The consuming data packet buffer is still at the same level; and the address packet buffer still keeps at a very low level most of the time, but reaches the maximum 2048 packets sometimes.

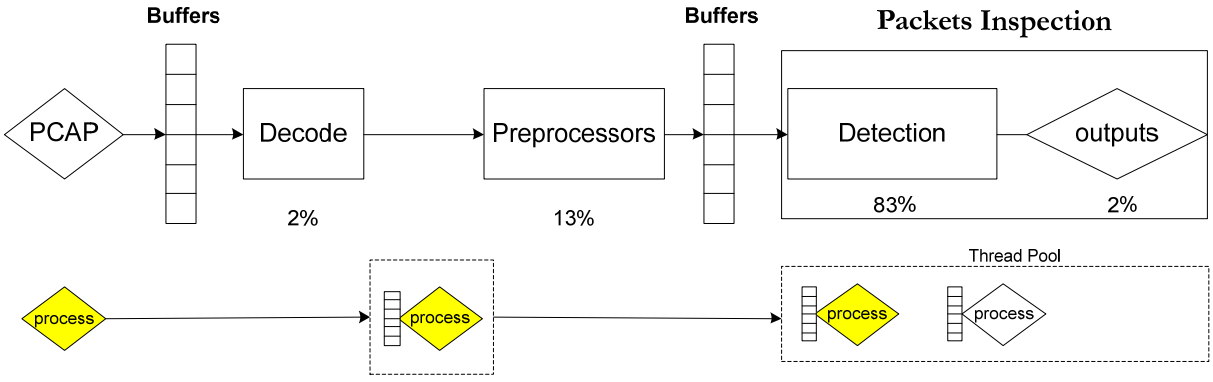


Figure 77: Snort test structure S02 (1+1+2).

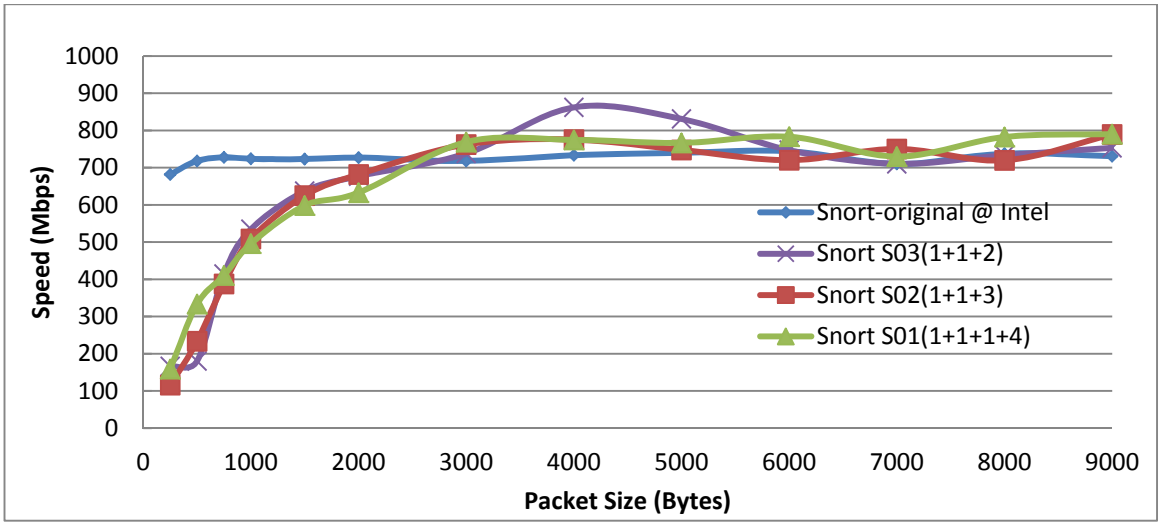


Figure 78: The inspection speed of Snort S02 (1+1+2)

6.6 PCAP Capture Buffer

Figure 78 can confirm that the performance limitation is the memory bandwidth. In the pipelined Snort, because an incoming packet needs to be accessed by multiple different processes, all packets need be buffered. The LibPCAP is only designed for single sequential processes. It can capture a single packet each time. The captured packet will be copied from the OS socket to a single packet buffer allocated by LibPCAP. Snort has to copy the packet from the single packet buffer into its own multiple packet data circular buffer. A schematic is shown in Figure 79.The two memory copy processes by two separate programs waste some time and memory bandwidth.

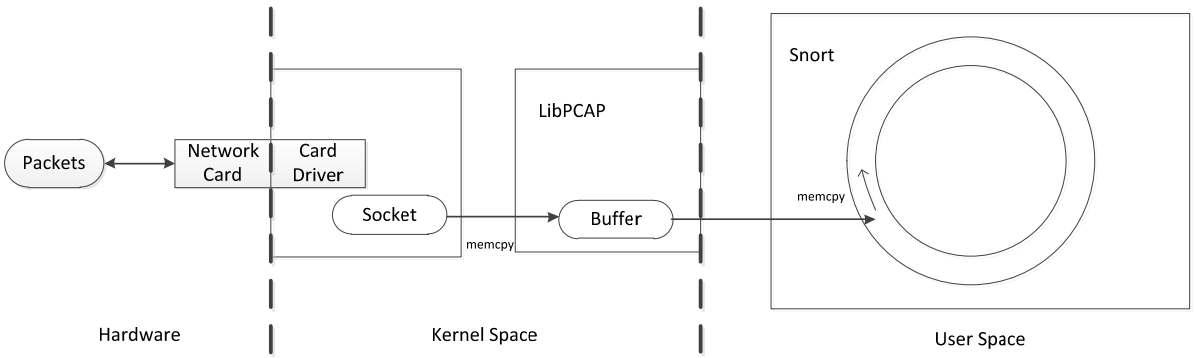


Figure 79: Packet buffer schematic of LibPCAP.

A solution to the above issue is combining these two packet buffers in LibPCAP and external programs into one buffer location. The packet buffer can be defined and allocated by the external program, such as Snort. When Snort calls LibPCAP to capture a packet, it can pass the buffer address to it. Then LibPCAP can copy the packet data and save its header into the received address from Snort directly. A schematic is shown in Figure 80.

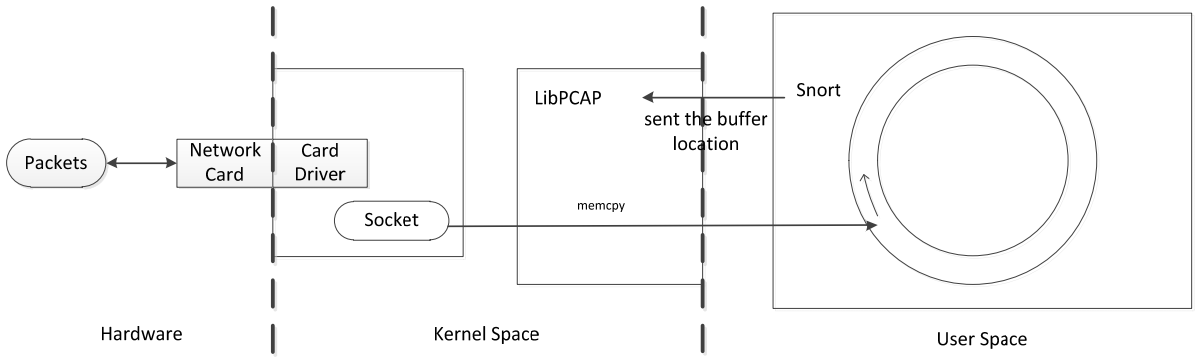


Figure 80: An external packet buffer schematic of LibPCAP.

In the implementation, some extra functions are added to the existing LibPCAP. A list of primary functions is shown in Figure 81.

```
<pcap.c>:
/* main function can be called externally to capture a single packet by LibPCAP.
  pcap_t *p: pcap handler.
  struct pcap_pkthdr *pkt_header: the address of packet header allocated externally.
  const u_char **pkt_data: the address of packet data allocated externally.          */
int pcap_next_exbuf(pcap_t *p, struct pcap_pkthdr *pkt_header, const u_char **pkt_data);

<pcap-linux.c>:
/* Initial packet capture parameters in Linux system for external buffer*/
pcap_t * pcap_create_exbuf(const char *device, char *ebuf);
/* Capture a packet in Linux system, and save it to the external buffer*/
static int pcap_read_packet_exbuf(pcap_t *handle, pcap_handler callback, u_char *userdata)
{
    packet_len = recvfrom(handle->fd, bp + offset, handle->bufsize - offset,
                          MSG_TRUNC, (struct sockaddr *) &from, &fromlen);
}

<savefile.c>:
/* Read a packet from a file, and save it to the external buffer*/
int pcap_offline_read_exbuf(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
{
    status = sf_next_packet(p, h, p->buffer, p->bufsize);
}
```

Figure 81: A list of primary functions for the external buffer in LibPCAP.

6.7 Test of LibPCAP external Buffer

With implementation and using the external buffer feature on LibPCAP, the same tests in section 6.5 have been carried out again. The comparisons of the overall packet inspection are shown from Figure 82 to Figure 84.

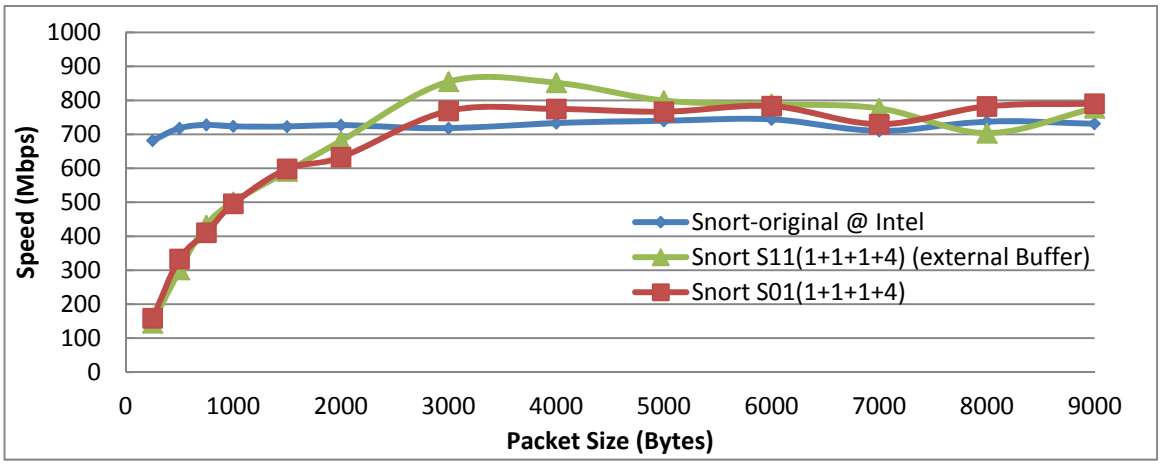


Figure 82: The inspection speed of Snort S11 (1+1+1+4).

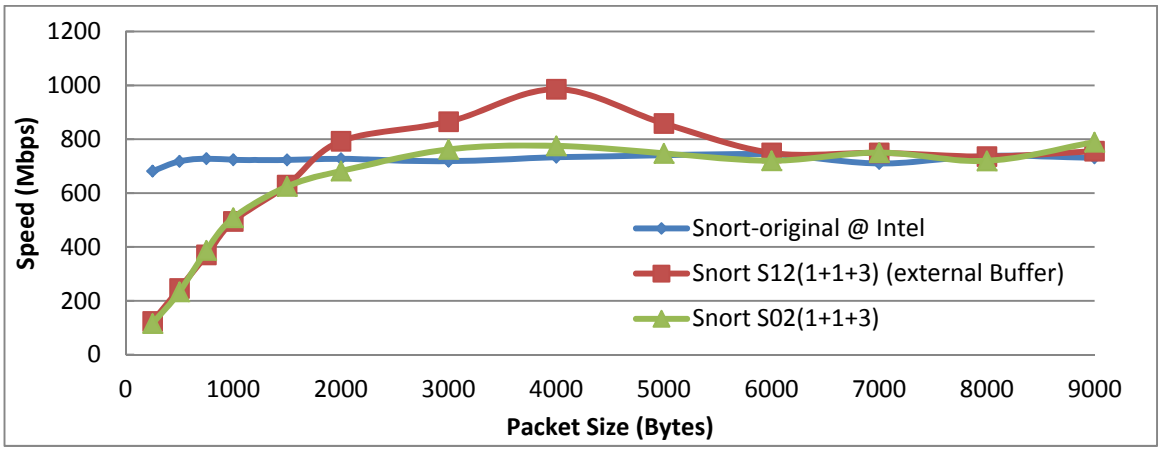


Figure 83: The inspection speed of Snort S12 (1+1+3)

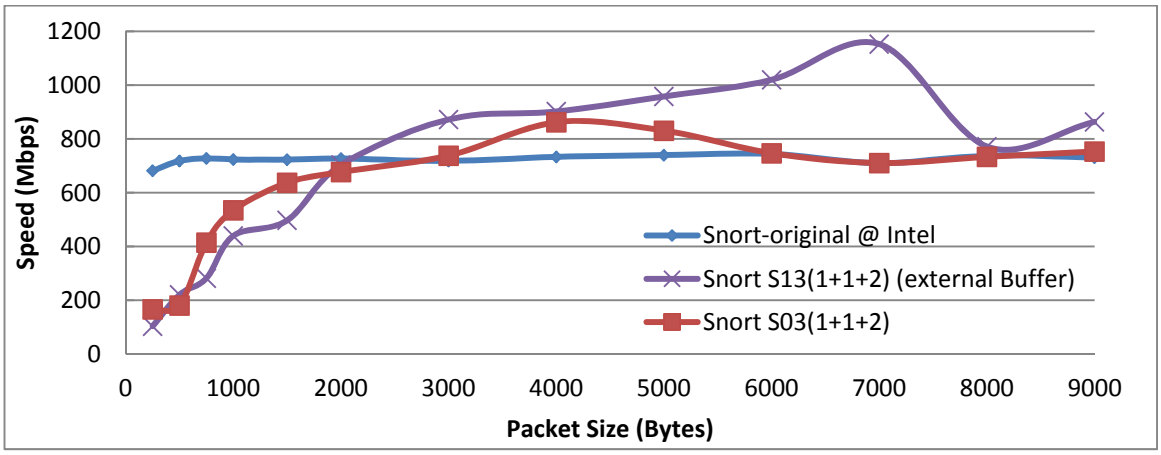


Figure 84: The inspection speed of Snort S12 (1+1+2)

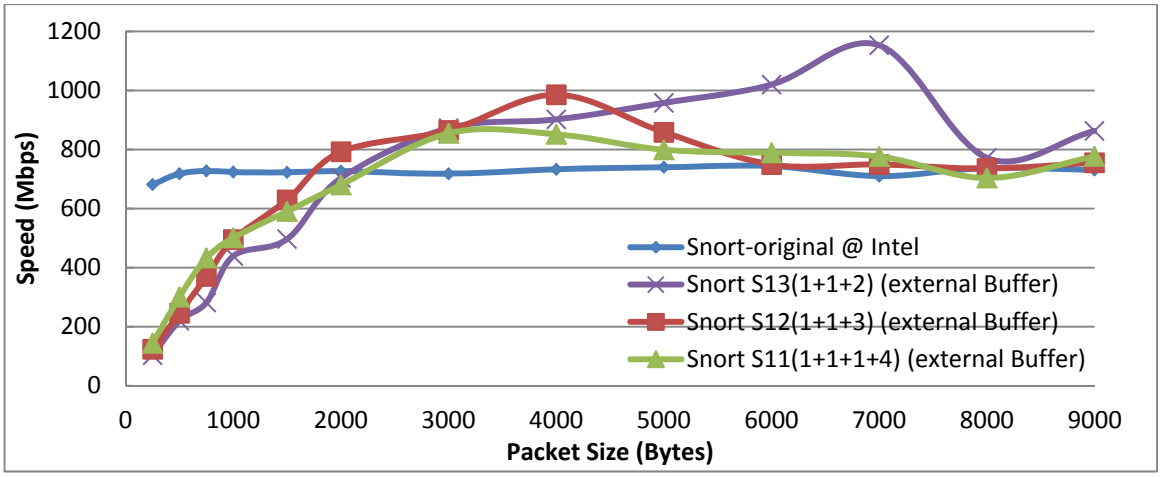


Figure 85: The inspection performance comparison using an external buffer in LibPCAP.

The above figures show that the overall performance of the packet inspection is improved when the size of packet is bigger than 2000 bytes by using an external buffer in LipPCAP. However, the performance does not change much when the size of the packet is below 2000 bytes or bigger than 8000 bytes. The performance is still confined to the memory bandwidth. A relatively shorter length of the pipeline and fewer processes can provide a better performance. The pipeline 1-1-2 can provide a maximum of about a 60% better performance than the original sequential Snort. With a bigger memory bandwidth, the performance of the pipeline should be able to increase greatly.

6.8 Results Review

Intel officially released a performance report on a multi-thread Snort in 2008. **[68]** In the report, Intel chose a similar packet pipeline and flow-pinning software architecture based on Snort 2.8.0.1. The main difference is that Intel implemented the structure by multi-thread; and the above Snort modules are implemented by multi processes with shared memory. Also, Intel only released the test result based on a 1 + 3 two-stage pipeline; the above tests are based on at least a three-stage pipeline. Intel provided test results for both offline (directly reading packets from the hard drive) and online and a real traffic online performance test had

been done by a hardware packet generator. However, owing to the limitation of the device, only offline tests are provided in the above tests.

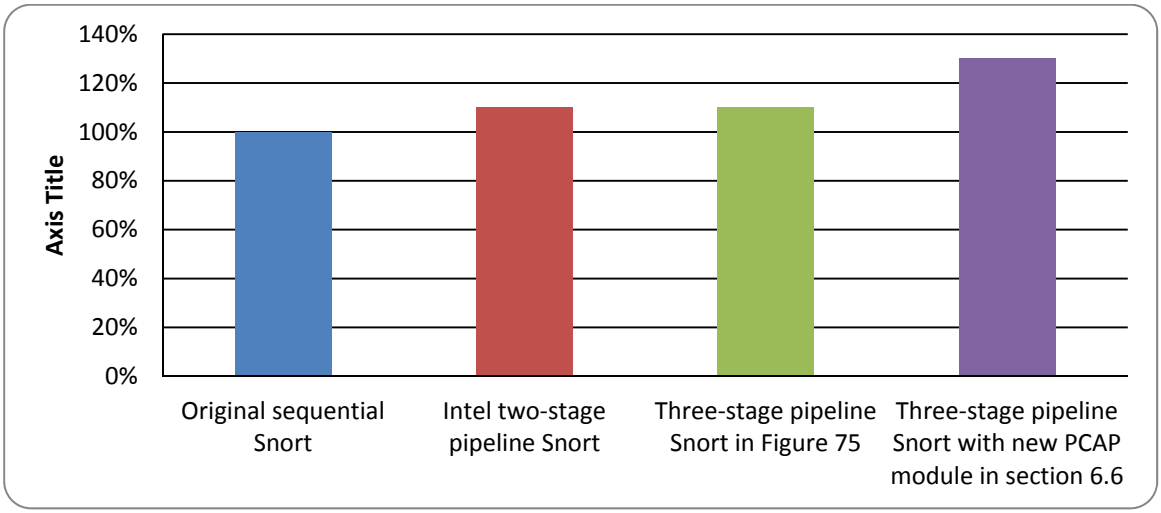


Figure 86: Snort 2.8.x offline test performance result for a 32KB packet.

Intel's pipeline module is very similar to the module shown in Figure 75. The result of the 32kB offline test matches the result shown in Figure 76. At the end of Intel's report, it declared that the bottleneck was on the packet capture library. The above Section 6.6 provides a new improvement on the PCAP module, which avoids another memory copy caused by the packet capture module. Figure 83 in Section 6.7 shows that the modified Snort with the external buffer PCAP module has a 30% better performance at the 32kB packet data. A performance comparison chart is shown in Figure 86.

In the above tests, all small packets with a size less than 1kB have a similar process speed in the number of packet per second (PPS). It is because these packets have a big overhead on the memory. The work load of different processes shows that all processes are waiting for "recvfrom" function in the Linux kernel, which is the new bottleneck of the system. The main job of the "recvfrom" function is reading packets between different computer hardware devices, and it contains some memory operations. Therefore, the bottleneck on the hardware is the bandwidth of the memory and bus.

6.9 Summary

Currently, the speed of the CPU is already very fast. In the multiple processors system, the bottleneck of the overall system performance moved from the CPU frequency to data acquisition speed, such as memory, cache and bus speed and bandwidth.

On a general purpose computer, normal packets with a size of less than 2000 bytes or bigger than 8000 bytes are unsuitable for packet level pipeline parallelization. A jumbo frame with a size between 2000 bytes and 8000 bytes can be parallelized in the packet level by a pipeline with a maximum of 60% performance improvement over the original Snort.

In addition, the external buffer feature of LibPCAP should also be used for packet level parallelization in other applications. It leaves the packet buffer in the user space, and the external program will have full control to access the buffer. If the memory bandwidth is allowed, it will be possible to perform parallelization in different ways.

At this point, there has been only a small amount of research into Snort parallelization on the general purpose processor. Most research has been performed on the FPGA or a special network processor. From the reported result, the overall performance of Snort on the FPGA and the network processor is better than on the general purpose computer.

IV. CONCLUSION

7 Conclusion

The main focus of this thesis is on some possible parallelization methods of the NIDS system in the general computer environment. The study is based on a famous open source NIDS system – Snort. The thesis has analyzed the structure and working mechanism of Snort and its packet capture API LibPCAP at code level. It then focused on the highest work load module in Snort – the pattern matching system, an overall pipeline structure of Snort, and a modification of LibPCAP.

For the pattern matching engine, some existing single and multiple pattern matching algorithms have been analyzed. Two parallelized pattern matching algorithms, the Reversed Bit-Parallelism algorithm and the Parallelized Aho-Corasick algorithm, have been designed and implemented. However, owing to the small granularity of the network packets, the pattern matching engine of Snort is unsuitable for parallelization.

In the overall function module level, a pipeline structure of Snort can be used to perform parallelization on a multi-core machine. In the pipeline system, all incoming packets are stored in a “data” circular buffer which can be accessed by all processes. To reduce the multiple memory copy from the operation system kernel socket to the packet buffer in the Snort, an external buffer packet capture feature has been added to LibPCAP. The inter-processes communication system in the pipeline has been carried out by shared memory. Both processes in the pipeline are connected by an “address” circular buffer. A very small message which contains the address of the packet in the data circular buffer will be passed in the middle of the pipeline between processes. The whole system can provide a maximum of

40% - 60% performance improvement over the official Snort system for jumbo frames on an Intel i7 computer. Unfortunately, the original sequential Snort still has a better performance for the packet with a size of less than 2000 bytes. Owing to the limited memory bandwidth, a big pipeline system does not help Snort to improve its performance, On an Intel i7 system, one processor for PCAP, one processor for the decoder and preprocessor and two processes for the inspection module give a better performance. More processes in the pipeline will introduce more overhead into the system.

At this point, Snort parallelization on FPGA still provides the best performance. The general purpose processor has the best flexibility and if the memory bandwidth can be increased in the future, it will also be a good option.

8 Future Work

In the tests, the “address” circular buffer usage keeps at a very low level most of the time, which means the PCAP may be the slowest part in the pipeline. The external packet buffer feature only gives the sequential LibPCAP the ability to support a multiple threads multiple processes user space program. Therefore, some more research can be carried out on the parallelization of LibPCAP to support multiple threads multiple processes natively.

In addition, another possible change to the LibPCAP is adding a packet classification feature. Chapter 4 of this thesis shows that the BSD Packet Filter is the kernel of the LibPCAP. It has a very fast speed to filter the traffic. A high speed packet classification feature would be easy to add on top of the BSD Packet Filter.

TCP protocol works on a session base and all packets in the same stream have an order dependency. Snort has a stream plugin in the pre-processors. A packet classification performing after the pre-processor session does not require handling of the packet stream issue. If a packet classification is to be carried out before the pre-processor of Snort, such as in the LibPCAP, a classification method needs to be designed.

Finally, there will be more potential to perform deep packet inspection parallelization with multiple sensors, in the network area.

9 References

[1] Rebecca Copeland, “Clark School Study: Hackers Attack Computers Every 39 Seconds”. 2007

[2] Martin Roesch, Official Snort website, http://www.snort.org.

[3] Anant Agarwal and Markus Levy, "Going multicore presents challenges and opportunities", 2007

[4] Paul Teich, "Multi-Core Processor Technology: Maximizing CPU Performance in a Power Constrained World", AMD.

[5] Henriok, “A schematic overview of the Blue Gene/L supercomputer”, http://en.wikipedia.org/wiki/File:BlueGeneL_schema.png

[6] Henriok, “Schema ASIC Blue Gene/L”, http://en.wikipedia.org/wiki/File:Blue_Gene_L_ASIC.png

[7] Anant Agarwal, Markus Levy, "The KILL Rule for Multicore", DAC 07 Proceedings of the 44th Annual Design Automation Conference (2007), ACM, 2007

[8] “International standard ISO OSI model”, http://en.wikipedia.org/wiki/OSI_model

[9] D. L. Schuff and V. S. Pai, "Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface," 2007 IEEE.

[10] “MAC layer PDU becomes physical layer SDU”, http://en.wikipedia.org/wiki/Protocol_data_unit

[11] Neil Matthew, Richard Stones. (2007). "Beginning Linux Programming". Wiley Publishing, Inc.

[12] Herlihy, Maurice; Moss, J. Eliot B. (1993). "Transactional memory: Architectural support for lock-free data structures". The 20th International Symposium on Computer Architecture (ISCA).

[13] Morry Katz, "PARATRAN: A transparent transaction based runtime mechanism for parallel execution of Scheme", MIT LCS, 1989

[14] Nir Shavit and Dan Touitou. "Software Transactional Memory". The 14th ACM Symposium on Principles of Distributed Computing, August 1995.

[15] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. "Software Transactional Memory for Dynamic-Sized Data Structures". the Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC). July 2003.

[16] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, J. McDonald, “Parallel Programming in OpenMP”. Morgan Kaufmann, 2000.

[17] Huang, Z.; Sun, C.; Cranefield, S.; Purvis, M."View-based consistency and its implementation", Cluster Computing and the Grid, 2001.

[18] Z. Huang, W. Chen, “Revisit of View-Oriented Parallel Programming”. Seventh IEEE International Symposium on Cluster Computing and the Grid, 2007.

[19] Zhiyi Huang, Chengzheng Sun, Stephen Cranefield, and Martin Purvis, View-based Consistency and its Implementation, in Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2001), pp.74-81, IEEE Computer Society, Brisbane, May 2001.

[20] Huang, Z.; Purvis, M.; Werstein, P. "Performance Comparison between VOPP and MPI", Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005.

[21] Huang, Z.; Purvis, M.; Werstein, P. "Performance evaluation of view-oriented parallel programming", Parallel Processing, 2005. ICPP 2005.

[22] Huang, Z., Purvis M., and Werstein P., Performance Evaluation of View Oriented Parallel Programming. In Proc. of the IEEE International Conference on Parallel Processing (ICPP05), pp251-258, IEEE Computer Society (2005), Oslo.

[23] Huang, Z., Chen, W., Purvis, M., Zheng, W., and Werstein, P., VODCA: View-Oriented, Distributed, Cluster-Based Approach to Parallel Computing, International Transactions on Systems Science and Applications, ISSN 1751-1461, Vol. 2, No 4, pp. 333-345, Feb 2007

[24] Zhang, J., Huang, Z., Chen, W., Huang, Q., Zheng, W., Maotai: View-Oriented Parallel Programming on CMT processors, in Proceedings of the 37th International Conference on Parallel Processing (ICPP08), IEEE Computer Society (2008), Portland, Oregon, USA.

[25] Leung, K.C., Huang, Z., Huang, Q., Werstein, P., Maotai 2.0: Data Race Prevention in View-Oriented Parallel Programming. In Proceedings of the Tenth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2009). pp., IEEE Computer Society (2009), Japan.

[26] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System", Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 207–216, 1995.

[27] Todd Brian, “Putting multicore processing in context”. EETimes Design, 2006

[28] Amdahl, Gene . "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" AFIPS Conference Proceedings (30), 1967.

[29] Gonzalo Navarro, Mathieu Raffinot, "Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences", Cambridge University Press, 2002

[30] Ricardo Baeza-Yates, “Approximate String Matching”, Center for Web Research, www.cwr.cl ,Depto. de Ciencias de la Computacion, Universidad de Chile, Santiago, CHILE

[31] Christian Charras, Thierry Lecroq, “Handbook of Exact String-Matching”, College Publications

[32] R.M. KARP, M.O. RABIN, "Efficient randomized pattern-matching algorithms", IBM J. Res. Dev, 1987

[33] Jeffrey E. F. Friedl, "Mastering Regular Expressions, 3rd Edition", O'Reilly, August 2006.

[34] D.E. Knuth, J.H Morris., V.R. Pratt, 1977, Fast pattern matching in strings, SIAM Journal on Computing

[35] Robert S. Boyer, J Strother Moore, "A fast string searching algorithm", Communications of the ACM, October 1977, Volume 20, Number 10

[36] R.N. Horspool, 1980, "Practical fast searching in strings, Software - Practice & Experience"

[37] C. Allauzen, M. Crochemore, and M. Raffinot. "Efficient experimental string matching by weak factor recognition". In Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching.

[38] Ricardo Baeza-Yates and Gaston H. Gonnet, "A new approach to text searching", Communications of the ACM, October 1992/%1.35, No.10

[39] Alfred V. Aho, Margaret J. Corasick, "Efficient string matching - An aid to bibliographic search", Communications of the ACM, October 1975, Volume 18, Number 6

[40] S. Wu and U. Manber. "A fast algorithm for multi-pattern searching", Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[41] C. Allauzen and M. Raffinot. "Factor oracle of a set of words", Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

[42] Alam, M.S.; Javed, Q.; Akbar, M.; Rehman, M.R.U.; Anwer, M.B. “Adaptive load balancing architecture for SNORT”, Networking and Communication Conference, 2004. INCC 2004.

[43] Botwicz, J.; Buciak, P.; Sapiecha, P. “Building Dependable Intrusion Prevention Systems”, Dependability of Computer Systems, 2006

[44] Lih-Chyau Wuu, Sout-Fong Chen, "Building intrusion pattern miner for snort network intrusion detection system", IEEE 37th Annual 2003 International Carnahan Conference on Security Technology, 2003.

[45] Dharmapurikar, S.; Lockwood, J.W. "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems", Selected Areas in Communications, IEEE Journal on Volume: 24 , Issue: 10, 2006

[46] H. Liu. "Efficient Mapping of Range Classifier into Ternary CAM". In IEEE Symposium on High Performance Interconnects, Stanford, CA, August 2002.

[47] H. Song and J.W. Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In ACM International Symposium on FPGAs,Monterey, CA, February2005.

[48] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification using Extended TCAMs. In IEEE International Conference on Network Protocols. IEEE, 2003.

[49] R. Franklin, D. Carver, and B. L. Hutchings. "Assisting Net-work Intrusion Detection with Reconfigurable Hardware". In IEEE Symposium on Field-programmable Custom Computing Machines, Napa Valley, CA, April 2002. IEEE.

[50] R. Sidhu and V. K. Prasanna. "Fast Regular Expression Matching using FPGAs". In IEEE Symposium on Field Programmable Custom Computing Machines, Napa Valley, CA, April 2001. IEEE.

[51] Y. H. Cho and W. H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, April 2004. IEEE.

[52] Y. H. Cho and W. H.Mangione-Smith. Programmable Hardware for Deep Packet Filtering on a Large Signature Set. In First IBM Watson P=ac2 Conference, Yorktown, NY, October 2004. IBM.

[53] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In IEEE Hot Interconnects 12, Stanford, CA, August 2003. IEEE Computer Society Press.

[54] Papadopoulos, G.; Pnevmatikatos, D. "Hashing + memory = low cost, exact pattern matching". Field Programmable Logic and Applications, 2005.

[55] Weinsberg, Y.; Tzur-David, S.; Dolev, D.; Anker, T. "High performance string matching algorithm for a network intrusion prevention system (NIPS)". High Performance Switching and Routing Workshop, 2006

[56] Zhuojun Zhuang; Yuan Luo; Minglu Li; Chuliang Weng. "A Resource Scheduling Strategy for Intrusion Detection on Multi-core Platform". Network and Parallel Computing, 2008.

[57] Piyachon, P.; Yan Luo. "Efficient memory utilization on network processors for deep packet inspection". Architecture for Networking and Communications systems, 2006

[58] Lin Tan; Sherwood, T. "A high throughput string matching architecture for intrusion detection and prevention". Computer Architecture, 2005.

[59] Sourour, M.; Adel, B.; Tarek, A. "A Stateful Real Time Intrusion Detection System for high-speed network", Advanced Information Networking and Applications, 2007.

[60] A. F. Arboleda and C. E. Bedon, “Snort diagrams for developers, Version 0.2 alpha” Universidad del Cauca – Colombia, 14th April 2005

[61] The official web site of LibPCAP. “http://www.tcpdump.org”.

[62] Luis MartinGarcia, "Programming with Libpcap - Sniffing the network from our own application". Hakin9 Magazine. Issue 2/2008

[63] Cisco Systems, “Building Scalable Cusco Internetworks”, Volume 1 Version 3.0, Cisco Systems, 2006

[64] Rebecca Bace and Peter Mell, “Intrusion Detection Systems”, NIST Special Publication, 16 August 2001. http://csrc.nist.gov.

[65] Nen-Fu Huang, Chih-Hao Chen, Rong-Tai Liu, Chia-Nan Kao, and Chih-Chiang Wu, “On the design of a cost effective network security switch architecture”, Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE.

[66] Nen-Fu Huang, Chih-Hao Chen, Yang-Fang Huang, Yi-Hsuan Feng, Chia-Nan Kao, Hsien-Wei Hung, and Ming-Chang Shih, "A Scalable Architecture for High Available Security Switches", Communications, 2006. ICC '06. IEEE.

[67] Young H. Cho and William H. Mangione-Smith, "Deep Network Packet Filter Design for Reconfigurable Devices." ACM Transactions on Embedded Computing Systems (ACM TECS), 2007.

[68] Intel Corp., "Removing System Bottlenecks in Multi-threaded Applications." Intel Application Note, Order Number: 320631-001US, Sep 2008.