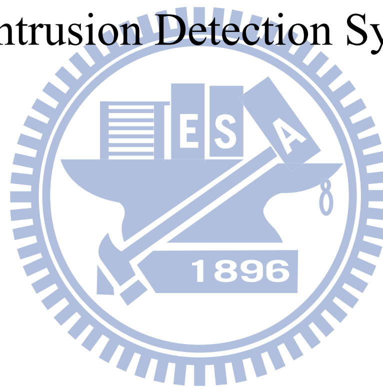
國 立 交 通 大 學

電機與控制工程學系

碩士論文

對於網路入侵偵測系統之功能平行化樣本比對演算法

A Function-Parallelism Pattern-Matching Algorithm for Network Intrusion Detection Systems



研 究 生：洪精佑

Student: Ching-You Hung

指導教授：黃育綸 博士

Advisor: Dr. Yu-Lun Huang

中華民國九十八年

Spring, 2009

對於網路入侵偵測系統之功能平行化樣本比對演算法

A Function-Parallelism Pattern-Matching Algorithm for Network Intrusion Detection Systems

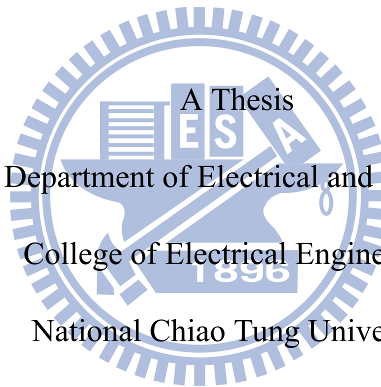
研 究 生：洪精佑　　Student: Ching-You Hung

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

國 立 交 通 大 學

電機與控制工程學系

碩士論文



A Thesis

Submitted to Department of Electrical and Control Engineering

College of Electrical Engineering

National Chiao Tung University

in partial Fulfill of the Requirements

for the Degree of

Master

in

Department of Electrical and Control Engineering

Spring, 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年

對於網路入侵偵測系統之功能平行化樣本比對演算法

學生：洪精佑　　　　　　　　　　指導教授：黃育綸　博士

國 立 交 通 大 學電機與控制工程學系（研究所）碩士班

摘　　要

在網路入侵偵測系統(NIDS)中，處理封包速度過慢的NIDS對於所保護的系統會有安全性上的漏洞。其中，樣本比對演算法佔著影響系統效能的關鍵性角色。在本篇論文中，我們試著分析目前NIDS中常見的樣本比對演算法，並且針對不同的演算法探討影響其效能的因素。我們發現，樣本群中最短樣本的長度對於Wu-Manber演算法有決定性的影響，當最短樣本的長度太短，會使其演算法效能過慢，另外同字首的樣本數量過多也會拖累比對的速度，而使得攻擊者可經由設計封包內容大量使用此字首拖累比對的速度，此稱為演算法攻擊(Algorithmic Attackes)。而對於Aho-Corasick演算法，長度短的樣本或是相同字首的樣本群卻可使比對速度快過一般的情況。因此，我們提出結合此兩種演算法的資料結構，並且透過功能平行化的方式將演算法對應到多核心系統中，可加速樣本比對的速度，並使得儲存空間在可接受的範圍。透過比較各個演算法的實驗，我們可得到使用功能平行化的比對演算法在雙處理器系統上比起原先的演算法最快可達2.2倍的效能。並且對於演算法攻擊有一定的防禦力。

A Function-Parallelism Pattern-Matching Algorithm for Network Intrusion Detection Systems

Student: Ching-You Hung Advisor: Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

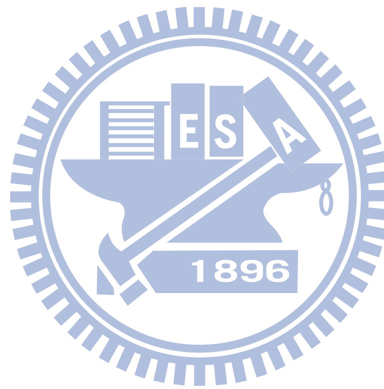
National Chiao Tung University

Abstract

Pattern-matching algorithms are the core of network intrusion detection systems (NIDS). The performance of a good pattern-matching algorithm hence dominates the processing time required for deep packet inspections. In this research, we discuss the factors that can affect the performance of a pattern-matching algorithm. Such factors include prefixes of rules and lengths of the longest rules in a ruleset. Previous work to improve the performance of matching patterns (Wu-Manber's and Aho-Corasick's algorithms) adopt either a hash table or finite automaton to store the rulesets. None of these algorithms considers the parallelization when running on multi-core systems. Herein, we propose a new pattern-matching algorithm for NIDS that can be easily adapted to multi-core systems. Our algorithm is composed of a search mechanism based on the function-parallelism approach and a composite data structure, combining the hash table and finite state machines. We conduct a series of experiments to show that our algorithm is 2.2 times faster than the Aho-Corasick algorithm and 1.21 times than Wu-Manber's in a dual-processor system.

謝誌

在這裡首先要感謝我的指導老師黃育綸博士，由於他的指導以及培育才能讓我產生出這篇論文。 另外也要謝謝實驗室的大家，特別是黃詠文學長熱心幫忙，才能讓這篇論文順利完成。 也很謝謝我的家人，由於他們的體諒我才能夠無後顧之憂地努力完成我的學業。 在兩年的碩士生涯中，謝謝學長姐、同學以及學弟妹們的幫助，讓我的知識與專業能力有長足的成長。除了課業上，在生活以及休閒上，有了這些人一起歡笑一起出遊，使得這兩年的生活更加豐富。 謝謝RTES Lab的大家。



Contents

摘要 i

Abstract ii

謝誌 iii

Table of Contents iv

List of Figures vi

Chapter 1 Introduction 1

1.1 Background 1

1.2 Contribution 2

1.3 Synopsis 3

Chapter 2 Related Work 4

2.1 SNORT 4

2.2 Pattern Matching Algorithms 6

2.2.1 Boyer-Moore Algorithm 7

2.2.2 Wu-Manber Algorithm 8

2.2.3 Aho-Corasick Algorithm 10

2.3 Analysis 11

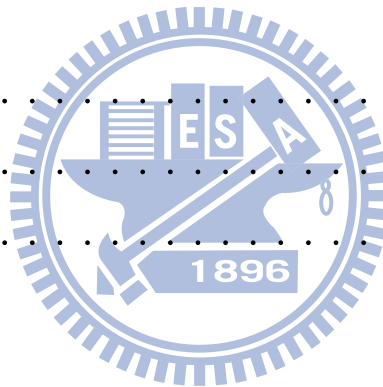
2.3.1 Space Complexity 12

2.3.2 Time Complexity 13

2.4 Discussion 14

2.4.1 Length 14

2.4.2 Prefix 15



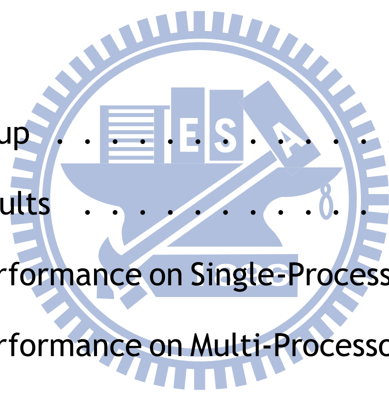
Chapter 3 Function-Parallelism Pattern-Matching Algorithm 16
3.1 Composite Data Structure 16
3.2 Function-Parallelism Search Mechanism 18
3.3 Computational Efficiency 19

Chapter 4 Implementation 23
4.1 FPPM Detection Engine 23
4.2 Case Study 24
4.2.1 Single-Processor System 25
4.2.2 Dual-Processor System 26
4.2.3 Multi-Processor System 27

Chapter 5 Experiments 28
5.1 Experimental Setup 28
5.2 Experimental Results 29
5.2.1 Exp#1: Performance on Single-Processor NIDS 30
5.2.2 Exp#2: Performance on Multi-Processor NIDS 30
5.2.3 Exp#3: Number of Cache Misses 31
5.2.4 Exp#4: Algorithmic Attack 32
5.2.5 Exp#5: Storage Cost 33
5.3 Discussion 34

Chapter 6 Conclusion and Future Work 35

References 36



List of Figures

2.1 The process flow of Snort 5

2.2 Tables used in the Wu-Manber algorithm 9

2.3 Example of a syntax tree 10

2.4 Example of a DFA in the Aho-Corasick algorithm 11

3.1 Combination of first two states in the DFA structure 17

3.2 Combination of the hash table in the Wu-Manber algorithm and the beginning state in the modified Aho-Corasick algorithm 18

4.1 The basic flow in the single-processor detection engine 25

4.2 The packet flow in the dual-processor detection engine 26

4.3 The packet flow in the multi-processor systems 27

5.1 Processing time of the algorithms in single processor 30

5.2 Processing time of the algorithms in dual processors 31

5.3 Number of cache misses in each algorithm 32

5.4 Processing time of each algorithm by the modified searching text 33

5.5 Comparison of the memory consumption of each algorithm 34

Chapter 1

Introduction

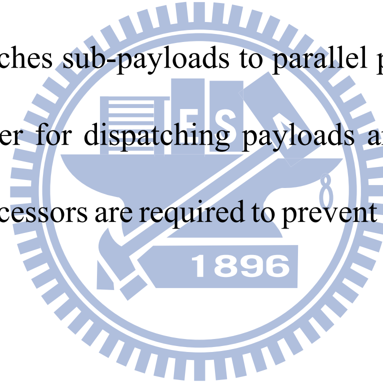
1.1 Background

The popularity of networking brings easy data access, but also makes system vulnerabilities exposed to the public. According to the statistics published by CERT [1], there are over 31,118 vulnerabilities being cataloged since 2004. Deploying shelters and alarms of attacks then becomes one of the major challenges to enhance the network security. In recent years, many researches focus on designing and developing network intrusion detection systems (NIDS) to monitor the network traffic and efficiently detect the malicious packets or flows. The existing NIDS systems [2][3][4] detect the malicious behaviors by comparing incoming packets with signatures and rulesets pre-stored in the database. Since some malicious codes may be concealed inside payloads, the NIDS systems adopt detection engines to check the content of payloads. A detection engine is a component of NIDS that takes data from the packet decoder and compares it against the rules configured in the NIDS. However, investigating a data payload is a computational consuming work and may congest the whole NIDS. A study of NIDS[5] shows that the detection engine takes about 75% of the total computing time of a NIDS. In other words, a delayed detection engine may expose the NIDS to a vulnerability of dropping packets with potential attacks.

The delay of a detection engine may come from the pattern-matching algorithms, which con-

tribute most in searching and detecting malicious signatures. Since different pattern-matching algorithms adopt different searching methods, some of them can get better efficiency with different types of patterns. Some of them take less time in searching patterns, while some others require less memory in storing rulesets. Many existing pattern-matching algorithms either intend to improve the searching speed or reduce the storage cost when running on a single-processor system, but none of them consider the parallelization when applied to a multi-processor system.

Trivially, a multi-processor system can improve the performance of detecting malicious signatures by simultaneously running the pattern-matching algorithms. However, by applying data-parallel approach to redesigning a pattern-matching algorithm may better improve the performance of a detection engine running on a multi-processor system. The data-parallel approach divides a payload and dispatches sub-payloads to parallel processors[6]. Such an approach requires an elaborate dispatcher for dispatching payloads and balancing data flows; otherwise synchronization between processors are required to prevent race condition when processing payloads.



1.2 Contribution

The novel contribution of this paper include:

- we propose a new function-parallelism pattern-matching (FPPM) algorithm for NIDS running on multi-core systems. The algorithm is composed of
  - a search mechanism based on the function-parallelism approach, and
  - a composite data structure, combining the hash table and finite state machines.

  The function-parallelism approach creates a queue for each processor for buffering payloads to be processed. Based on the function-parallelism approach, payloads are buffered

in the queues of processors. In such a design, since each processor has its own queue, it requires fewer efforts to serialize data.

- we implement a detection engine running the FPPM algorithm on both single- and multi-processor systems. For the realization of FPPM, several modifications were made to the new detection engine, including separating the buffer queues and enhance the parallelism of the original detection engine in multi-processor systems.

- we conduct a series of experiments to analyze the performance of the FPPM algorithm when running on single- and multi-processor systems. From the experimental results, we show that FPPM can reduce both the execution time and the memory space consumed by the detection engine.

E S A
1896

1.3 Synopsis

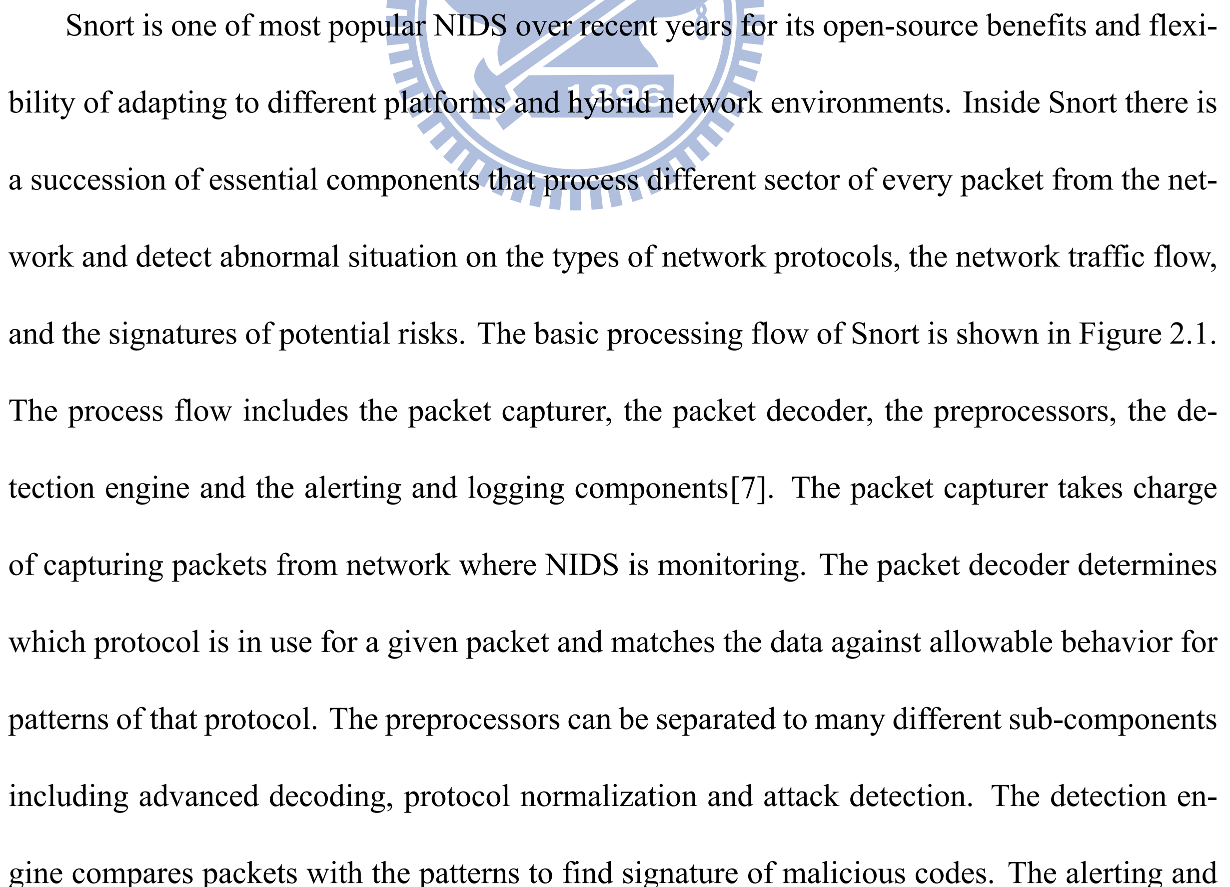
This thesis is organized as follows. We review the basic process flow of the NIDS and examine the pattern-matching algorithms used for the detection engine in Chapter 2. We also evaluate the performance of these pattern-matching algorithms in Chapter 2. Then, we propose the FPPM algorithms and study the efficiency of it in Chapter 3. The implementation of the new detection engine is described in Chapter 4 and the experimental results to confirm the performance of FPPM are shown in Chapter 5. In the end, we conclude the thesis in Chapter 6.

Chapter 2

Related Work

In this chapter, we examine the basic process flow of the NIDS and go deep into the most computing-consuming part -- the detection engine. We use Snort as an example because of its popularity and clarity. We further classify the pattern matching algorithms used in the detection engine of Snort. First, we dig inside the structure of Snort and dissect the functional components.

2.1 SNORT



Snort is one of most popular NIDS over recent years for its open-source benefits and flexibility of adapting to different platforms and hybrid network environments. Inside Snort there is a succession of essential components that process different sector of every packet from the network and detect abnormal situation on the types of network protocols, the network traffic flow, and the signatures of potential risks. The basic processing flow of Snort is shown in Figure 2.1. The process flow includes the packet capturer, the packet decoder, the preprocessors, the detection engine and the alerting and logging components[7]. The packet capturer takes charge of capturing packets from network where NIDS is monitoring. The packet decoder determines which protocol is in use for a given packet and matches the data against allowable behavior for patterns of that protocol. The preprocessors can be separated to many different sub-components including advanced decoding, protocol normalization and attack detection. The detection engine compares packets with the patterns to find signature of malicious codes. The alerting and

logging components take the appropriate actions in accordance with the signatures found by the detection engine or the preprocessors.

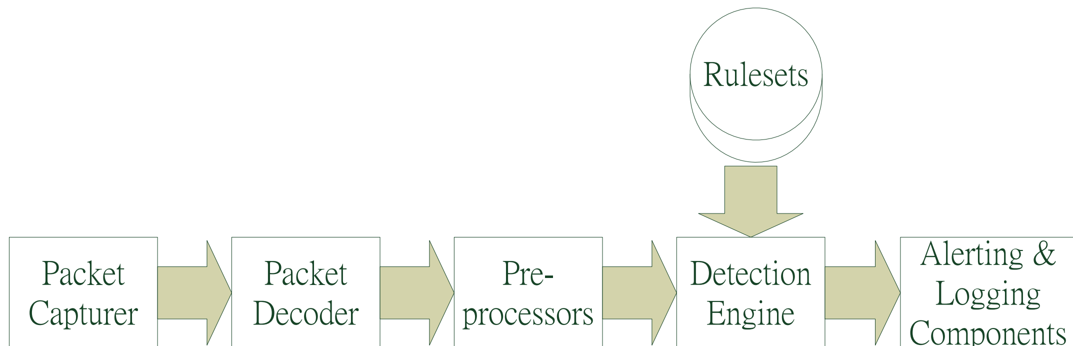


Figure 2.1: The process flow of Snort

The execution of a large number of patterns of the rulesets is inside the detection engine. In the detection engine, the rules used by the detection engine are separated into four rule groups: TCP, UDP, ICMP, IP. When processing a packet, the detection engine first checks the protocol. If the protocol is TCP, UDP, or ICMP, the detection engine checks the ruleset for that protocol; otherwise, it checks the IP ruleset. Then, the detection engine calls prmFindRuleGroup, which returns the appropriate pattern group based on source and destination ports in the packet, and passes the pattern-group matching function. Inside the matching function, the detection engine first checks any rules with uricontents. For each URI marked in the rule by the http_inspect preprocessor, the detection engine calls the setwise pattern engine. After checking each of the uricontents rules, then the regular content rules are checked in the same manner, last all of the rules without content.

When checking the uricontents and the content of a packet, the detection engine compares them with the patterns in the Snort rulesets by a specified pattern-matching algorithm. The users can specify the algorithms only at the configuration stage of Snort. Since the detection engine is a critical part of NIDS that affects performance of the overall system, it is important

to choose a suitable algorithm for different NIDS depending on the capacity of the system. An improper algorithm will debase the performance of the detection engine immensely. Methods of choosing a suitable algorithm depend on the hardware capacity such as computing power and memory space restriction. The type of pattern groups is also an important factor that affects the performance of the pattern-matching algorithms[8]. We will explain the impression in the following chapters.

2.2 Pattern Matching Algorithms

The procedure of a pattern matching algorithm can be separated into two stages, the pre-constructing and searching stage. At the pre-constructing stage, the pattern matching algorithm compiles the patterns for searching and adopts an optimized data structure to accelerate the searching speed. For multi-pattern matching algorithms, a group of patterns are enveloped to the specified structures, so it is capable of searching multiple patterns at a time. The data structure is stored in the memory for use at the searching stage.

At the searching stage, the algorithm takes the searching text as an input to the pre-constructed structure and find the matched patterns. Because the amount of the searching text is much ampler than the amount of patterns, the measurement of the performance of pattern matching algorithms is by the processing speed of the searching text at the searching stage. In addition, another measurement of these algorithms is the memory space consumed to construct the patterns to the specified structure at the pre-constructing stage. In this thesis, we use these two measurements as the indicators to the performance of the pattern matching algorithms.

Basically, pattern matching algorithms used in Snort can be categorized into two approaches, heuristic-based approach and automaton-based approach [9]. The heuristic-based approach

skips the dispensable characters to accelerate the searching speed and the automaton-based approach is based on deterministic finite automata (DFA) to process input payloads against the patterns. In Snort, The Boyer-Moore algorithm [10] and The Wu-Manber algorithm [11] process the pattern matching by heuristic-based approach and the Aho-Corasick algorithm [12] is an antique but efficient algorithm based on automaton-based approach. These algorithms are explained in more detail below.

2.2.1 Boyer-Moore Algorithm

The Boyer-Moore algorithm is a speedy pattern matching algorithm because of its characters skip ability. Two tables, the bad-character shift table and the good-suffix shift table, are built at the pre-constructing stage of the Boyer-Moore algorithm. The bad-character shift table records the minimum length can be skipped when searching character mismatches to the right-most character of the pattern. If the mismatching character of the searching text does not appear in the pattern, the next character being compared can be skipped the pattern length long. The good-suffix shift table records the minimum length can be skipped when the suffix of the pattern matches but a mismatching occur in the middle of the pattern. In Snort, a variant of the Boyer-Moore algorithm is actually used, which is known as the Boyer-Moore-Horspool algorithm or Horspool's algorithm which only applies the bad-character shift table.

Given a pattern pat of length m, the bad-character shift table is an array of 256 elements, which is 256 possible conditions of a one-byte character, with the initial value m. Then, the $pat[m]$-th element of the array is set to 0, the $pat[m-1]$-th element is set to 1, and so on until the $pat[1]$-th element is set to $m-1$.

At the searching stage with a searching text tex of length L, the algorithm first get the shift value by inputting $tex[i]$, where i is initialized as m, to the bad-character shift table. If the shift

value is not 0, means that the searching character is not matching to the last character of the pattern, and the shift value is added up to i and repeat this process. If the shift value is 0, which means a hit of the last character of the pattern, and $tex[i-1]$ will be compared with $pat[m-1]$ and keep comparing until a mismatch happens or the entire pattern is matching to the piece of the searching text. This procedure will be repeated until reaching the end of the text.

The Boyer-Moore algorithm or Boyer-Moore-Horspool algorithm has much good performance for the lengthy patterns but the drawback is that it can only perform on single pattern at a time. That is, if the database has n different patterns, the payload of each packet needs to be checked n times. In Snort, there are over 1000 patterns in a pattern group [13], using single-pattern matching algorithm makes the detection engine work inefficient due to only one pattern being searched at a time, so we need multi-pattern matching algorithm that can execute searching process with multiple patterns at a time.

2.2.2 Wu-Manber Algorithm

The Wu-Manber algorithm is a multi-pattern matching algorithm using the ideas of the Boyer-Moore algorithm. Like the Boyer-Moore algorithm, Snort actually uses a modified version of the Wu-Manber algorithm. For the simplicity, here we refer the modified Wu-Manber algorithm as the Wu-Manber algorithm.

At pre-constructing stage, the Wu-Manber algorithm maintains a shift table according to the first m characters of all patterns, where m is the minimum length of all patterns. The shift table of the Wu-Manber algorithm is similar to the bad-character shift table of the Boyer-Moore algorithm but based on 2 characters instead of just one. The Wu-Manber algorithm also builds a hash table that contains all the patterns by hashing first two characters and uses list structure to connect the patterns with the same hash value. For example, if the ruleset includes three patterns,

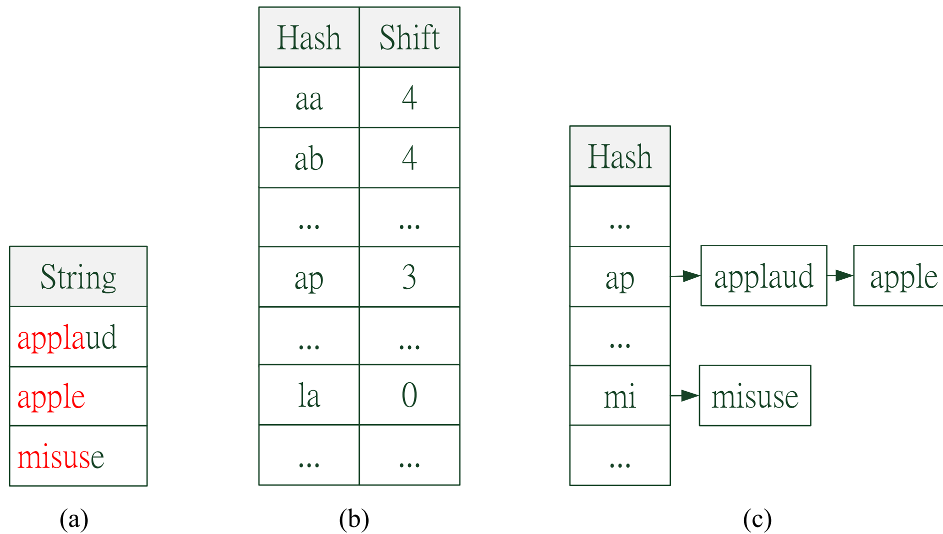


Figure 2.2: Tables used in the Wu-Manber algorithm: (a) Cutting the strings to minimum length (b) Shift table by 2-byte characters (c) Hash table by the first 2 prefix of patterns

apple, *applaud* and *misuse*.the shortest pattern is *apple*, whose length m is 5. Then the Wu-Manber algorithm takes the first 5 characters of all patterns to construct the shift tables, showing in Figure 2.2(a). The shift table is decided by two continuous characters in first m characters of all pattern. Taking string *pl* as a example, *pl* appears in the second word of *apple* and *appla* from the rightmost, so its shift value is 1, showing in Figure 2.2(b). Last, the Wu-Manber algorithm constructs hash table by the first two characters of all patterns. In this example, *apple* and *applaud* will be in the same list due to the same prefix, Figure 2.2(c).

At the searching stage, the algorithm works like the Boyer-Moore algorithm, using the $(i-1)$-th and i-th, where i is initialized as m, character of the searching text as an input to find the shift value of the shift table. If the shift value is not 0, then i is added up the shift value up and repeats the above operation. If the shift value is 0, which means these 2 characters on searching is matched to $(i-1)$-th and i-th character of some patterns, The Wu-Manber algorithm next calls the hash table with the $(i-m-1)$-th and $(i-m)$-th character of the searching text, gets the head of the list, which links the patterns with the same 2 prefix, and searches through the list finding the matching patterns.

2.2.3 Aho-Corasick Algorithm

The Aho-Corasick algorithm is another famous multi-pattern matching algorithm used in Snort, which construct a DFA structure using every character of each pattern as a link to next state. At the pre-constructing stage, the beginning state is assigned first. Then the states are linked to the previous state repeatedly according to each character of the pattern and form a syntax tree. Figure 2.3 shows a syntax tree using *apple* and *applaud* as an example. The final state is a cycle with boldface. After building the syntax tree, the left links of each state are linked to the corresponding states. A state, which has 256 possible next conditions of a one-byte character, has 256 links to next states, and some links point to the states with duplicate prefixes. Figure 2.4 shows the DFA from the syntax tree of *apple* and *applaud*. For the clearness, we omit the links which point to the beginning state.

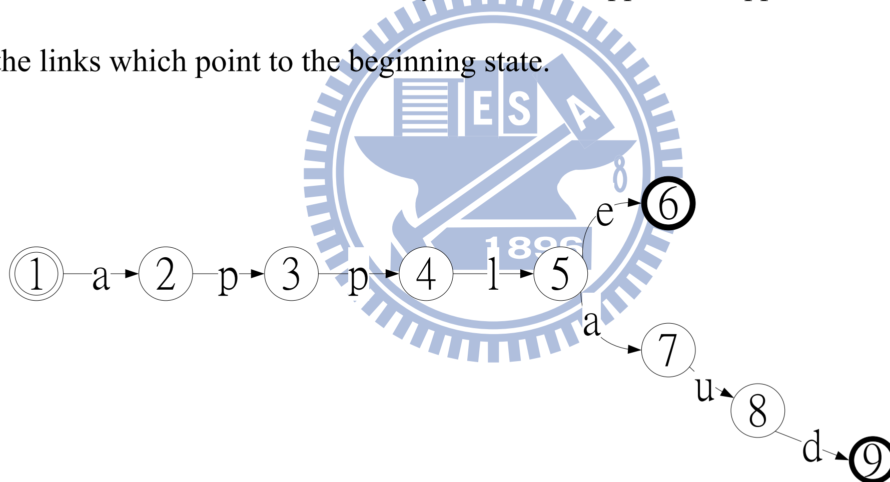


Figure 2.3: Example of a syntax tree

When at the searching stage, the Aho-Corasick algorithm starts at beginning state and each character of the searching text makes a state transition to the next state. Once current state is a final state, it means a pattern is found and a message will be sent to the detection engine for further inspection. The state transition will be executed until the last character of the searching text is checked.

One obvious drawback of the Aho-Corasick algorithm is that the memory space consump-

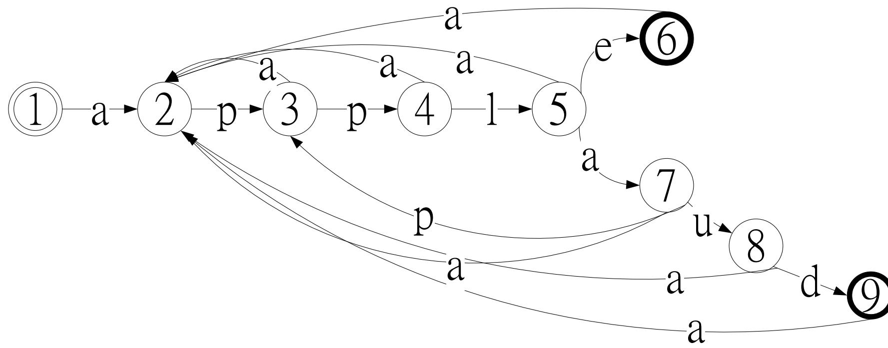


Figure 2.4: Example of a DFA in the Aho-Corasick algorithm

tion of the DFA structure is considerable. There are some researches [14][15] proposed to reduce the heavy memory consumption of the Aho-Corasick algorithm, but the addition of operations used for compress and uncompress the structure also increases the execution time. The total memory consumption of the DFA structure will be evaluated more accurately in the next section.

There is another searching algorithm in Snort, the SFK algorithm, which searches the matching patterns using byte-trees. The SFK algorithm is for specially low memory computing environments and the searching speed is largely slower than the other two algorithms. We assume the our computing environments are morden personal computers or even network computers with sufficient memory space so we choose to omit the usage of the SFK algorithm.

2.3 Analysis

For the multi-pattern matching algorithms introduced above, we evaluate the space complexity and the time complexity by different characteristics of the pattern groups. First, we need to build the performance formulas of the algorithms.

We first define the variable we need in evaluation of the performance of different algorithms.

pt is the size of an address pointer, *var* is the size of an integer variable, *Npat* is the total number of the patterns, *Lpat* is the average length of the patterns and *Spat* is the size of a pattern element, which records the necessary items of a pattern, like pattern ID, length, depth and a pointer to the next pattern.

2.3.1 Space Complexity

The hash table of the Wu-Manber algorithm stores the lists hashing by first two characters of each pattern and therefore it needs 256×256 address pointers to different lists. In all of the lists, the sum of total pattern elements is included, which increases $Npat \times Spat$ to the total memory consumption. In addition, the shift table also needs 256×256 integer variables to indicate the shift length by 2 character input. The total memory usage is:

$$256^2 \times pt + 256^2 \times var + Npat \times Spat \tag{2.1}$$

Also, the Aho-Corasick algorithm maintains a DFA where each state with 8-bits data that can branch to 256 different results, which means when adding a state to the DFA, 256 address pointers will be occupied to store next 256 different states from the original state. The total memory consumption of a DFA is:

$$256 \times pt \times states + Npat \times Spat, \tag{2.2}$$

where $states = Npat \times Lpat$. By comparing these two equations, we can conclude that the memory consumption of the Aho-Corasick algorithm expand more quickly than of the Wu-Manber algorithm when adding new patterns. And for the pattern groups with large average length, the Aho-Corasick algorithm needs much more memory space to store the DFA structure.

2.3.2 Time Complexity

The evaluation of the time complexity of different pattern matching algorithm is much more complicated. To effectively estimate the total time consumption by calculating the most time-consuming part of these algorithms, we simplified the equations to only measure the number of memory accesses. The measurement of number of memory accesses may not precisely tell the total time of execution by the pattern matching processes, but it can reveal a tendency of each algorithm when bringing into different pattern groups.

For the number of memory accesses in the Wu-Manber algorithm, we also need to define that SV is the average shift value, $P_{hash}(i)$ is the probability of hash table hit with the amount of patterns i in the list. We can calculate the number of memory accesses with the searching text of length L:

$$\frac{L}{SV} \times \sum_{i=1}^{max} P_{hash}(i) \times i, \tag{2.3}$$

where max is the maximum number of patterns of the lists in the hash table. One of the risks of the Wu-Manber algorithms is that the attackers can elaborately arrange the payload of packets to shorten the average shift value SV and increase the probability of $P_{hash}(i)$ with large amount of patterns i, and therefore slow down the speed of the pattern matching process. For the pattern group with short SV, the attackers only have to increase $P_{hash}(max)$ to efficiently stick the system.

The Aho-Corasick algorithm bases on the DFA, whose searching time is roughly linear to the length of payload, $O(n)$, but we further evaluate the number of memory accesses of a DFA. We define $P_{current}(i)$ is that the probability of the current state pointing to the i-th state and $P_{next}(i)$ is the probability of the i-th state changing to a different state when inputting the next searching character, and we get the number of memory accesses with the searching text of length

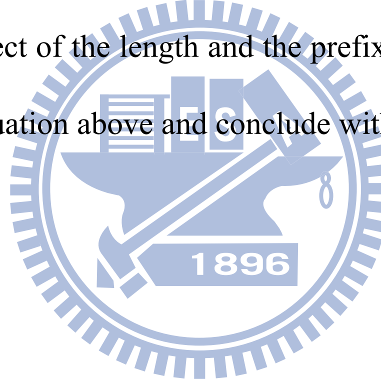
L:

$$L \times \sum_{i=1}^{states} P_{current}(i) \times P_{next}(i), \tag{2.4}$$

where $states = Npat \times Lpat$ and $\sum_{i=1}^{states} P_{current}(i) = 1, 0 \leq P_{current}(i) \leq 1$. We can predict the probability is concentrated in the shallow states in the average case, which means $P_{current}(i)$ is a rather large number for state i is close to the beginning state. And For a diverse pattern group, the probability of changing state of the beginning state $P_{next}(1)$ is near to 1 with large amount of patterns.

2.4 Discussion

Now we discuss the effect of the length and the prefix in the pattern groups on both algorithms according to the evaluation above and conclude with some suggestion on separating the pattern groups.



2.4.1 Length

The Wu-Manber algorithm is a fast multi-pattern searching algorithm but the searching speed depends heavily on the minimum length of all the patterns which is the maximum length the payload can be skipped. The minimum length of the pattern group has a direct effect on the average shift value SV because SV can never exceed the minimum length of the pattern group. That means longer minimum length in pattern group makes better searching performance to the Wu-Manber algorithm.

There is not obvious effect of the pattern length on the Aho-Corasick algorithm, but for the same number of patterns but with narrow average pattern length, the Aho-Corasick algorithm can reduce the total number of $states$, consequently increasing the cache hit rate and the searching

speed.

It is a good way to put lengthy pattern groups into the Wu-Manber group for enhancing the searching speed and put narrow pattern groups into the Aho-Corasick group for reducing the memory consumption. For only one pattern group, we can separate the pattern group by the length and construct each pattern sub group by each algorithm to get better performance than the original algorithm.

2.4.2 Prefix

Also, a large number of patterns with the same prefixes may cause a problem for the Wu-Manber algorithm that links all the patterns with the same first two characters to a list structure. When the input of searching text hit the valid hash value of the hash table, a large time cost will be paid to search patterns through the list. However, in the Aho-Corasick algorithm, the patterns with the same prefixes means can reduce the former states for construction of DFA, then reducing the memory consumption.

If there are hybrid lengthy and narrow patterns with the same prefixes, using the Aho-Corasick algorithm to construct the pattern group into the DFA structure can prevent the Wu-Manber from the algorithmic attacks.

Chapter 3

Function-Parallelism Pattern-Matching Algorithm

We proposed a new structure called function-parallelism pattern-matching (FPPM) algorithm that combined the automaton-based and heuristic-based approach to fasten the pattern matching speed in multi-processor systems. Two main kinds of algorithms mostly used in the detection engine of Snort, the Aho-Corasick algorithm of the automation-based algorithms and the Wu-Manber algorithm of the heuristic-based algorithms, are applied to construct the new algorithm. We first modified these algorithms to be able to combine with each other to reduce the memory utilization. Then, we separate the pattern group into two sub pattern groups by an important character of the patterns, the minimum length of the pattern group, each sub pattern group constructed by either one of these modified pattern matching algorithms, to enhance the performance of each algorithm by function-parallelism approach. Also, we further evaluate the performance of our new pattern matching algorithm in the last of the chapter.

3.1 Composite Data Structure

As mentioned in previous chapter, when the minimum length of the pattern group exceeds a threshold, The Wu-Manber algorithm has a better performance than the Aho-Corasick algorithm in average case. For this reason we separate the pattern group into two sub groups by the length and use the Wu-Manber algorithm to search the sub group with lengthy patterns. With the sub

group with short patterns, we use a modified Aho-Corasick algorithm to do the pattern matching process. In this modified Aho-Corasick algorithm, we construct the pattern group into a DFA structure and combine the former two states of the DFA into one state, as figure 3.1 shows. Because when state skipping in the DFA with short patterns, the current state often points to the former states, the benefits of combining the former states of the DFA are to reduce the memory fetching frequency at the former states, and to combine with the hash table of the Wu-Manber algorithm more easily.

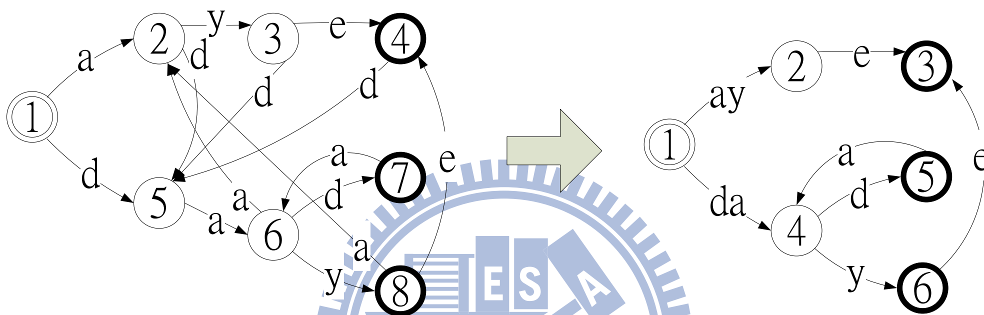


Figure 3.1: Combination of first two states in the DFA structure

Next, we combine the hash table of the Wu-Manber algorithm and the beginning state of the modified Aho-Corasick algorithm into the prefix table. We first separate the one-byte patterns from the pattern group need for pattern-matching process, and construct a one-byte table to store them by lists structure. For the left patterns, we construct a prefix table to link all of them. Each element in the prefix table contains an address to the list in hash table or the next state in a DFA, as figure 3.2 shows. Each item in the prefix table contains either structure dynamically according to *length_threshold*, which is the minimum length of the pattern group searched by the Wu-Manber algorithm. First, the pattern group is separated by the first two characters, that means there are 256^2 sub groups, each sub group with the same two prefixes. Then, *length_threshold* decides each sub group to one of the two groups, the Aho-Corasick group and the Wu-Manber group, by comparing the minimum length of the sub group with it. If the minimum length of

the sub group is over *length_threshold*, then this sub group is assigned to the Wu-Manber group, or assigned to the Aho-Corasick group if not. Algorithm 1 shows the pre-constructing stage process of the FPPM algorithm.

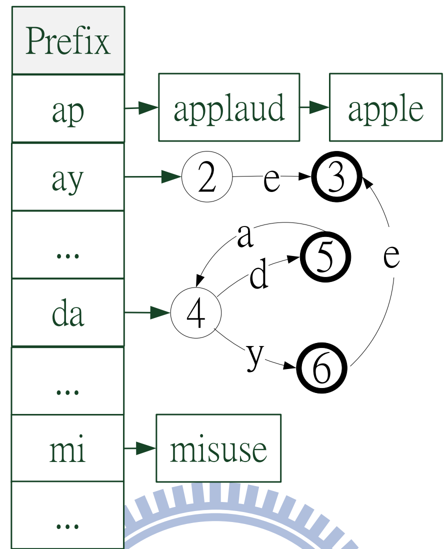


Figure 3.2: Combination of the hash table in the Wu-Manber algorithm and the beginning state in the modified Aho-Corasick algorithm

3.2 Function-Parallelism Search Mechanism

At the searching stage, both approaches, heuristic-based and automaton-based, are separated by the function-parallelism approach to do each searching process. In the heuristic-based approach, the bad-character shift table works just like in the Wu-Manber algorithm. When the bad-character shift table returns 0, the FPPM algorithm first checks the prefix table using first two characters as an input by retreating *(m-1)* characters back from the current searching character, where m is the maximum length can be shifted, and see if this prefix group belongs to the Wu-Manber list structure. If true, then searching process through the list will be started. If false, it means no element corresponding to this prefix and continues searching patterns with next characters. Algorithm 2 shows the heuristic-based part of the FPPM algorithm.

Algorithm 1 Pre-Construction Stage Process of FPPM Algorithm

for each $pattern[i]$ **do**
  PUT $pattern[i]$ to $prefix[\ pattern[i][0]\ ][\ pattern[i][1]\ ]$
end for
for each $prefix[j][k]$ **do**
  COUNT the minimum length of patterns in $prefix[j][k] = minlen$
  if $minlen \leq length_threshold$ **then**
    PUT $prefix[j][k]$ to AC_DFA
  else
    PUT $prefix[j][k]$ to WM_LIST
  end if
end for
for every $pattern[i] \in$ AC_DFA **do**
  CONSTRUCT $pattern[i]$ to the modified DFA
end for
for every $pattern[i] \in$ WM_LIST **do**
  CONSTRUCT $pattern[i]$ to the Wu-Manber hash table
end for
COMBINE the modified DFA & the Wu-Manber hash table

In the automaton-based approach, the prefix table in FPPM stands for the first state. The FPPM algorithm first checks if the prefix group belongs to the DFA structure with first two characters of the searching text. If true, then the current state is pointed to the hitting state and searches for the next state by the next character, as the Aho-Corasick algorithm does. If false, the FPPM algorithm skips one character and then checks the prefix table with two characters next to the skipped character. The prefix table will be checked again and again until the input hits the DFA group. Also, for every input of the searching text, the automaton-based part will check the one-byte table to see if there are one-byte pattern hits. Algorithm 3 shows the automaton-based part of the FPPM algorithm.

3.3 Computational Efficiency

The structure of the FPPM algorithm combines the DFA and hash table and therefore consumes the memory space between the Aho-Corasick algorithm and the Wu-Manber algorithm.

Algorithm 2 Search Stage Process of the heuristic part in the FPPM Algorithm

Input $text$ with length len
$minlen \leftarrow$ length_threshold
$i \leftarrow minlen - 1$
while $i < len$ **do**
  if $ShiftTable[\ text[i-1]\ ][\ text[i]\ ] \neq 0$ **then**
    $i \leftarrow i + ShiftTable[\ text[i\text{-}1]\ ][\ text[i]\ ]$
  else
    if $prefix[text[i\text{-}minlen\text{+}1]][text[i\text{-}minlen\text{+}2]] \in$ WM_LIST **then**
      for each $pattern[j] \in prefix[text[i\text{-}minlen\text{+}1]][text[i\text{-}minlen\text{+}2]]$ **do**
        COMPARE $pattern[j]$ to $text + (i - minlen + 1)$
        if Matched **then**
          CALL Matched Function
        end if
      end for
    end if
    $i \leftarrow i + 1$
  end if
end while

Algorithm 3 Search Stage Process of the automaton part in the FPPM Algorithm

Input $text$ with length len
$i \leftarrow 0$
$CurState \leftarrow 0$
while $i < len$ **do**
  SEARCH One-Byte Table with $text[i]$
  if $CurState = 0$ **then**
    if $prefix[text[i]][text[i+1]] \in$ AC_DFA **then**
      $CurState = prefix[text[i]][tex[i+1]]$
      $i \leftarrow i + 1$
      SEARCH One-Byte Table with $text[i]$
    end if
  else
    $CurState \leftarrow State[CurState].next[text[i]]$
  end if
  if $State[CurState] = final$ **then**
    CALL Matched Function
  end if
  $i \leftarrow i + 1$
end while

Because the data structure of the FPPM algorithm utilizes the hash table and shift table, the memory consumption of FPPM is more than of the Wu-Manber algorithm. The FPPM algorithm separates the short patterns to the DFA and combines the front two states to the prefix table. Thus it consumes much less memory compared with the DFA with the same pattern group. According to eq. 2.1 and eq. 2.2, the total memory consumption of the FPPM algorithm is:

$$256^2 \times pt + 256^2 \times var + 256 \times pt \times AC_states + Npat \times Spat, \tag{3.1}$$

here AC_states means the total number of states constructed from the Aho-Corasick group. In the Aho-Corasick group, because the average length and the amount is less than in the original pattern group, AC_states is much less than $states$ in eq. 2.2.

Another benefit of the FPPM algorithm is to reduce the amount of memory accesses, therefore reducing the execution time. For the heuristic part of FPPM, compared with the Wu-Manber algorithms in eq. 2.3, the number of memory accesses is proportional to $\frac{L}{SV}$. through isolating the lengthy patterns to the Wu-Manber group, the minimum length of the Wu-Manber group is larger than the original minimum length, making sure that the original average shift value $SV_{original}$ and the new shift value SV_{new} are $SV_{origianl} \leq SV_{new}$ for the same searching text. In addition, through decreasing the amount of patterns, the probability of prefix table hit $P_{prefix}(i), i > 0$ can also be reduced, compared with $P_{hash}(i)$ in eq. 2.3.

For the automaton part of the FPPM algorithm, the equation of the amount of memory accesses is the same as in eq. 2.4, but the probability of pointing to the beginning state $P_{current}(1)$ is larger than the original. Besides, the probability of the beginning state changing to a different state $P_{next}(1)$ become much less in our case. Next, we prove that the amount of memory accesses of the automaton part is less than that of the original Aho-Corasick algorithm for the same pattern

group. We separate the original equation in eq. 2.4 to:

$$L \times (\sum_{i=1}^{N} P_{current}(i) \times P_{next}(i) + \sum_{i=N+1}^{states} P_{current}(i) \times P_{next}(i)), \tag{3.2}$$

where $state2 \sim stateN$ are next to the beginning state. Now we prove that the total amount of memory accesses in the FPPM algorithm is less than in the Aho-Corasick algorithm. By proving that the $New[P_{current}(1) \times P_{next}(1)] \leq \sum_{i=1}^{N} P_{current}(i) \times P_{next}(i)$ and the remainders stays the same, the above result can be proven. We first define that $P_{next-j}(i)$ is the probability of the i-th state changing to the j-th state, and we can get:

$$\begin{aligned}
&\sum_{i=1}^{N} P_{current}(i) \times P_{next}(i) \\
&\geq [\sum_{i=1}^{N} P_{current}(i)] \times [\sum_{i=2}^{N} P_{next-i}(1) \times P_{next}(i)] \\
&(\because P_{next}(1)\ is\ the\ smallest\ in\ P_{next}(i)\ and\ \sum_{i=2}^{N} P_{next}(i) = 1) \\
&\geq [\sum_{i=1}^{N} P_{current}(i)] \times \{\sum_{i=1}^{N} P_{next-i}(1)[P_{next}(i) - P_{next-1}(i)]\} \\
&= New[P_{current}(1) \times P_{next}(1)]
\end{aligned}$$

For the one-byte patterns, although the number of memory accesses is proportional to the searching text length L, the cache miss rate of accessing the one-byte table is near to 0 because the table size is rather small compared with the prefix table and the DFA, so we omit the effect of accessing the one-byte table.

The FPPM algorithm can reduce the amount of cache misses and the performance of FPPM increases along with the expanding pattern groups. We will show the experimental comparison with the original algorithms in chapter 5.

Chapter 4

Implementation

After constructing two pattern matching approaches in the FPPM algorithm, we next adapt them to the multi-processor systems, and for the single-processor systems, we also find a way to install the FPPM algorithm to it. It is important to carefully deploy the FPPM algorithm to the system because the poor deployment will reduce the total performance of the pattern matching process. We also discuss the deployment to different types of systems and explain the data flow in detail in this chapter.

4.1 FPPM Detection Engine

In the implementation of the FPPM algorithm, we use the source codes of the detection engine in Snort2.6.1 to generate the kernel of FPPM. Two source code files, *acsm.h* and *wmw.h*, are the kernels of the Aho-Corasick algorithm and the Wu-Manber algorithm separately in the detection engine of Snort. There are two important modules, the pre-constructing function and the searching function, inside each of these files. We combine the kernels of different algorithms in Snort and reserve the trivial codes like capitalizing the searching text and calling the matched functions if matched.

We also modify the detection engine of Snort to fit in with the function-parallelism request. The original detection engine of Snort is not adapted to the multi-processor systems and the only way to implement the original detection engine is by data-parallelism approach. The performance by this way may be depressed because of the synchronization efforts. In the modified

detection engine, we separate the different searching parts of the FPPM algorithm to new threads and we bind these threads to each processor to clear the context switch of different threads. The utilization of threads is to apply them to the multi-processor systems easily[16]. We also construct a queue for each processor to remove the mutual exclusion of accessing the input packets.

In this implementation, we provide the kernel of the FPPM algorithm, that includes:

- the pre-constructing function of the FPPM algorithm that combines the data structure of two different algorithms,
- the searching function of the FPPM algorithm that execute searching processing by the function-parallelism approach,
- and the information function that shows the patterns and memory information of the FPPM algorithm.

We also provide the new detection engine that applies multi-thread conformable to the multi-processor systems and makes a fine integration of FPPM. Next, we discuss the implementations to different types of the NIDS.

4.2 Case Study

We design different processing flows of the detection engine depending on how many processors the system has and explain the benefits of each processing flow that takes advantage of the FPPM algorithm. The concept behind the processing flow is to abate the synchronization efforts and reduce the probability of time block of the mutual exclusion. The details in the types of systems may be unique, but the performance of the detection engine can increase by applying this concept.

4.2.1 Single-Processor System

For the single-processor system, which can only execute an instruction at a time, the two pattern matching parts in the FPPM algorithm are lined up to a sequential way. When a packet comes in, the detection engine first checks the type and the input and output port of the packet. According to these information, the detection engine finds the pattern group that includes all signatures the packet may conceal. After picking up the pattern group, because the FPPM algorithm has two sub pattern groups and either group may be an empty group, the detection engine further checks if there is a need to process the heuristic part or the automation part to save the redundant time. Figure 4.1 shows the basic flow of the detection engine of single-processor system.

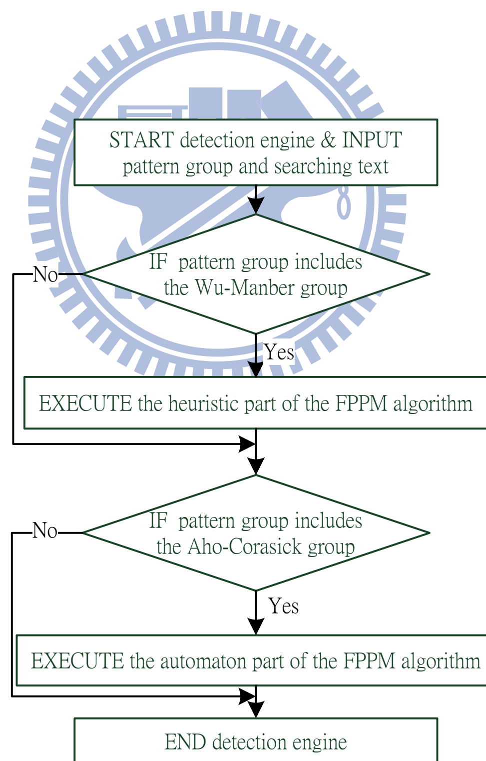


Figure 4.1: The basic flow in the single-processor detection engine

4.2.2 Dual-Processor System

In the dual-processor system, the FPPM algorithm can allot the heuristic and automaton part to different processors to reduce cache miss rate and enhance the searching speed. For each processor, we build different queues to buffer the packet payload needs to do deep packet inspection. The benefit of using separating queues is that the detection engine can avoid using mutual exclusion to prevent repeating the pattern matching process to the same packet. Waiting for mutual exclusion may waste a lot of computational energy. When a packet is processed by the pre-processors, the next step is sent to the detection engine for deep packet inspection. For the new detection engine using the FPPM algorithm, the packet payload will be copied or dispatched, according to the sub pattern group valid or not, to the queues of the processors where the pattern matching process is running with the specified approach and pattern group. Figure 4.2 shows the structure of the dual-processor detection engine.

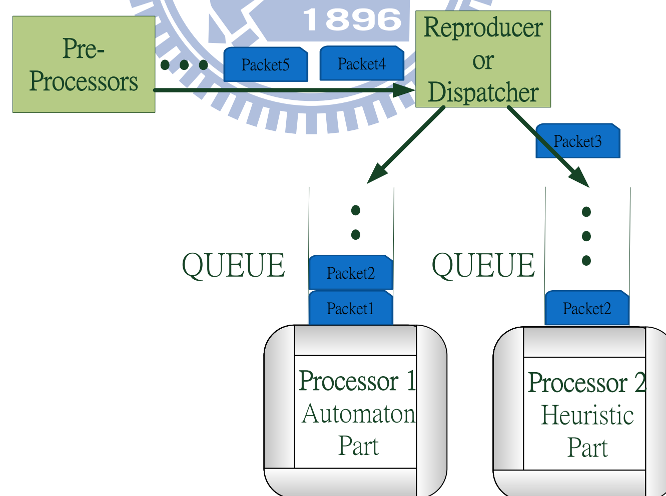


Figure 4.2: The packet flow in the dual-processor detection engine

1896

4.2.3 Multi-Processor System

The multi-processor system here refers to the system with multiples of two processors. This is also a common example of the multi-processor system. In the multi-processor system, we can bind the structure of the dual-processor system to any two processors (a FPPM set) in the multi-processor system, as Figure 4.3 shows. In addition to the "R or D" manager (in charge of reproduction or dispatch) in each set of processors, there is a dispatcher to allot every incoming packet to different FPPM set to balance the workload of each set.

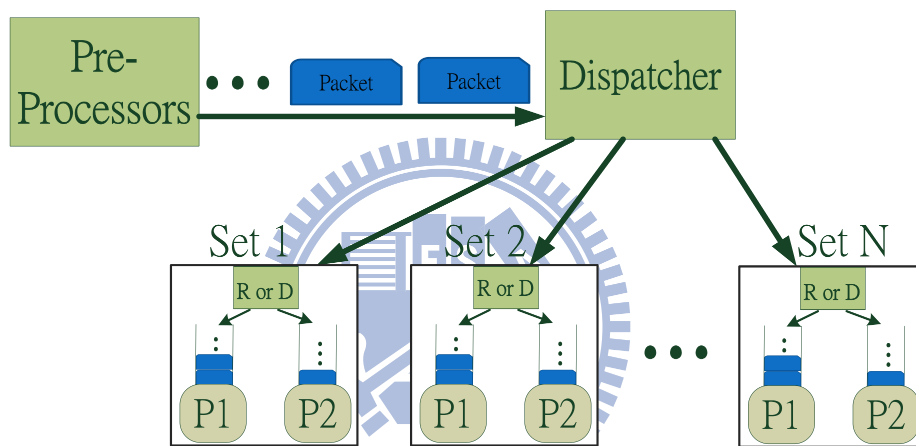


Figure 4.3: The packet flow in the multi-processor systems

Chapter 5

Experiments

In this chapter, we compare the performance of different pattern matching processes between the FPPM algorithm and the original algorithms, the Aho-Corasick algorithm and the Wu-Manber algorithm, running separately on a single-processor and multi-processor system. We use different number of randomized patterns as input and test different indicators of each algorithm. And then we analyze the result to confirm the evaluation in the previous chapters.

5.1 Experimental Setup

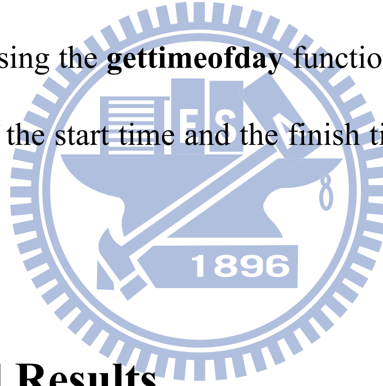
In our experimental platform, we set up a personal computer with a quad-processor CPU, 2G random access memory, installed with Linux operating system. The hardware and software specifications are shown below.

Component	Specification
CPU	Intel Core2 Quad Q6600 @ 2.40GHz
Level 1 cache	32KB (Date) & 32KB (Instruction)
Level 2 cache	4096 KB 16-way associative
RAM	2 GB
OS	Linux 2.6.24

Both the Aho-Corasick and Wu-Manber algorithms are distilled from the source code of Snort2.6.1 and compile by gcc 4.2.4[17]. The FPPM algorithm is modified from these two algorithms to fit the details described in the previous chapters. The searching text is generated from random

8-bit characters, and the data size is 170 MB. To make the searching text more like the actual data used in the network, we separate them to 20k packets by the length from 200 to 1500 bytes. The patterns are also randomly generated from the same random numbers and the pattern length is ranged from 1 to 40. The choise of the *length_threshold* is by simulation of the searching time of a little piece of random data with different values to check what value of *length_threshold* makes the best performance of the FPPM algorithm. The value of *length_threshold* is various with different pattern groups but the range is almostly from 5 to 7.

For the pure test, we first affiliate the OS routines and the applications to the first core of CPU, and bind the algorithms to the other cores of CPU to eliminate the influence of context switches and interrupt service routines. We record the total processing time of each algorithm by getting the system time using the **gettimeofday** function, which is a high-resolution timing function. The subtraction of the start time and the finish time will get the accurate processing time in our experiments.



5.2 Experimental Results

In the experiments, we first compare the searching time of the pattern-matching algorithms in single processor and dual processors to see the differences between the original sequential way and the function-parallelism approach. The number of cache misses is also an important factor that affect the searching time of the algorithm and we simulate the cache misses to show a tendency with the growing pattern group. We also experiment the worst case of each algorithm by modifying the searching text to measure the searching time when the algorithmic attacks take place. Last, we measure the memory consumption of different data structures constructed by each algorithm.

5.2.1 Exp#1: Performance on Single-Processor NIDS

In the first experiment, we measure the time consumption of each algorithm in processing the searching text in single processor. To process both parts of the FPPM algorithm, the automaton and heuristic, we use a sequential way described in the previous chapter. Figure 5.1 shows the result by different numbers of patterns. We can conclude that as the pattern group grows up, the FPPM algorithm can reduce the processing time by properly separating the pattern group to well fit algorithm. Of the processing speed with 10k patterns, the FPPM algorithm is 1.9 times faster than the Aho-Corasick algorithm and 1.18 times than the Wu-Manber algorithm.

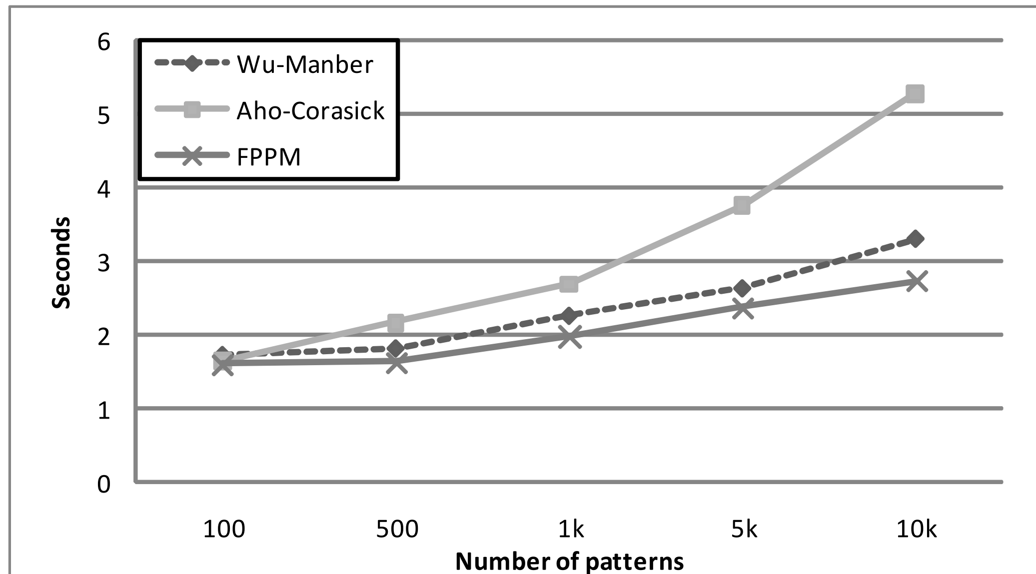


Figure 5.1: Processing time of the algorithms in single processor

5.2.2 Exp#2: Performance on Multi-Processor NIDS

The next experiment is to test these algorithms in multiple processors. We use two of the quad-processor CPU to test our algorithm, each processor with a 4MB L2 cache. Each processor with its own cache can reduce the competition for the limited cache resources and make the experiment more robust. For the Aho-Corasick and the Wu-Manber algorithm, we use data-

parallelism approach to apply them on the dual processors. And we bind each part of the FPPM algorithm to different processors of the CPU and measure the final processing time. Figure 5.2 shows the experimental result. With 10k patterns, the processing speed of the FPPM algorithm in dual processors is 2.2 times of the Aho-Corasick algorithm and 1.21 times of the Wu-Manber algorithm. With function-parallelism approach, the performance of the FPPM algorithm is better than the original algorithms because comparing to a bulky data structure, separating to small groups can reduce the frequency of distant memory accesses and hence reduce the cache miss rate. For a multi-processor system with individaul caches, using function-parallelism can reduce the duplicate cache misses of the same data.

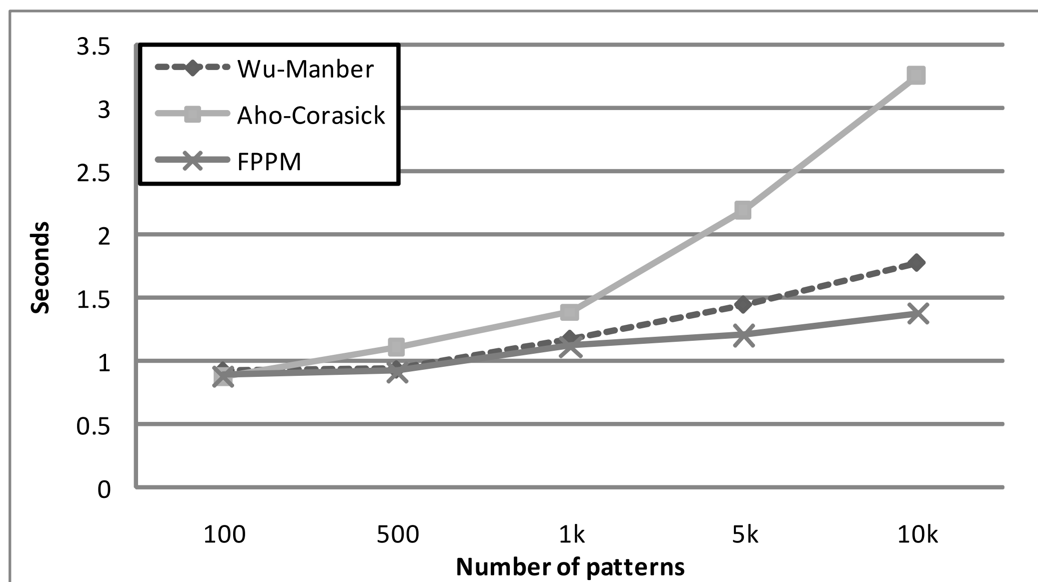


Figure 5.2: Processing time of the algorithms in dual processors

5.2.3 Exp#3: Number of Cache Misses

After showing the processing time in the single- and multi-processor system, we next calculate the number of cache misses to confirm the evaluation discussed in the previous chapters. At this experiment, we use a powerful memory management simulator *valgrind* [18], that can simulate the L1 and L2 cache miss in the IA-32 systems. An L1 cache miss costs as twenty

times of CPU circles as An L2 cache miss does. We summed up the scaled number of L1 cache misses and the number of L2 cache misses to indicate the total time cost of cache miss penalty. As figure 5.3 shows, as the pattern group grows up, the cache miss number of FPPM is more gently rising than the others.

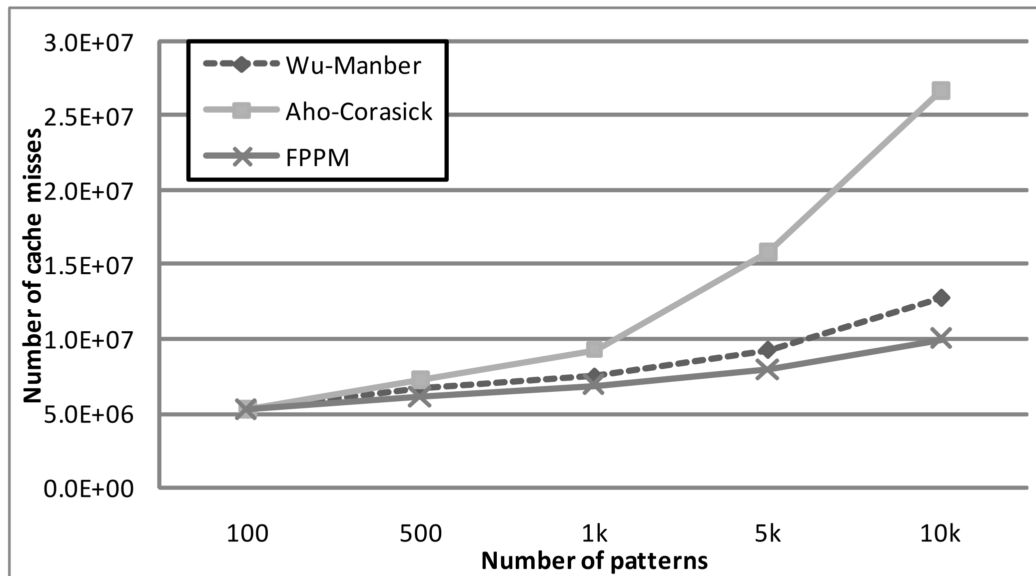


Figure 5.3: Number of cache misses in each algorithm

5.2.4 Exp#4: Algorithmic Attack

As mentioned in the previous sections, the Wu-Manber algorithm may suffer from the algorithmic attacks. In this experiment, we modified the searching text to abundantly hit the hash value with a list that includes the most patterns to congest the Wu-Manber algorithm. Figure 5.4 shows that the searching speed of the Wu-Manber algorithm is greatly reduced. For the Aho-Corasick algorithm, the searching time is rather decreased with the modified searching text because with a great number of duplicate characters, the memory accesses to the DFA can focus on the same states and reduce the frequency of data switch in cache. Therefore, the FPPM algorithm can hold out the searching speed by separating the patterns with the same prefixes to the Aho-Corasick group by $length_threshold$ and lengthening the shift length of the heuristic

part to reduce the number of comparison. .

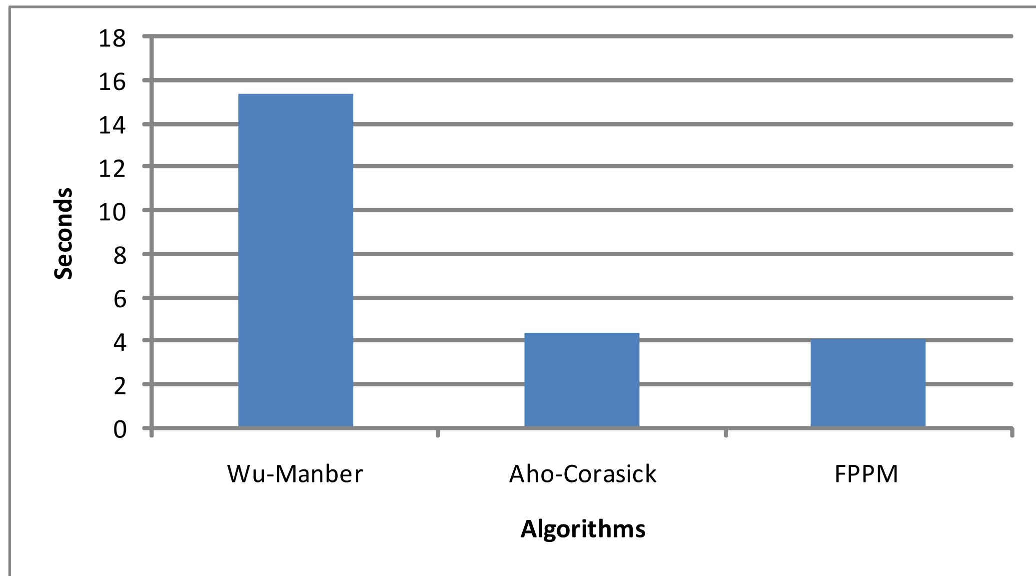


Figure 5.4: Processing time of each algorithm by the modified searching text

5.2.5 Exp#5: Storage Cost

Last, we compare the memory consumption of each pattern matching algorithm. Unlike the Aho-Corasick algorithm, in which the memory consumption is linear to the pattern number and the average length of the pattern group, the Wu-Manber algorithm uses the hash table and shift table to record the pattern elements, therefore the memory consumption nearly fixed to the table size as the pattern group grows up. The FPPM algorithm applies the structures of the Wu-Manber and the Aho-Corasick algorithm so the memory consumption is between these two algorithms and is greatly reduced from the structure of the DFA in which the memory consumption is proportional to the number of states. Figure 5.5 shows the memory consumption of each algorithm. Due to the separation of the lengthy patterns into the Wu-Manber group, the memory consumption of FPPM expands slowly when the pattern group grows up.

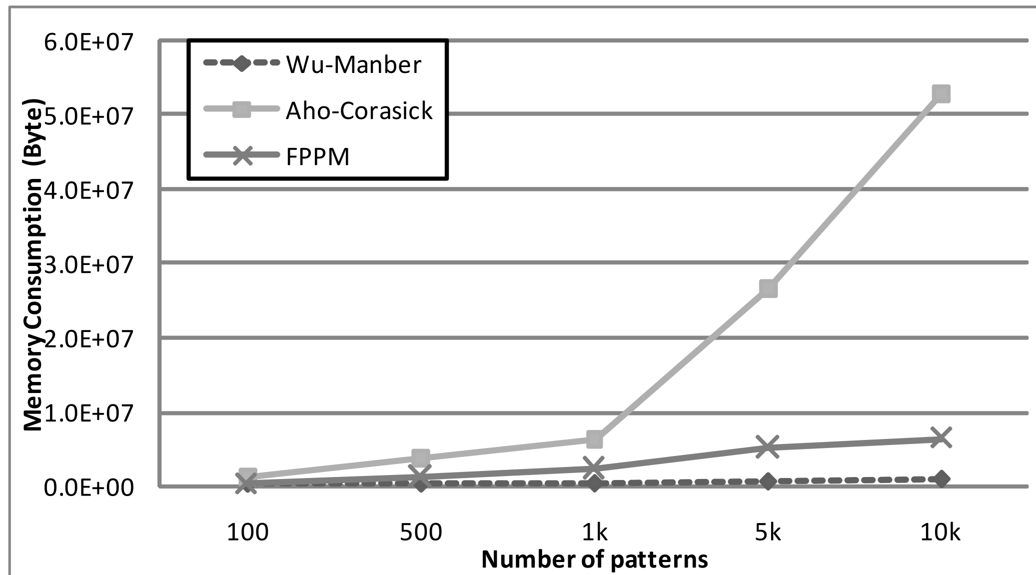


Figure 5.5: Comparison of the memory consumption of each algorithm

5.3 Discussion

We summarize these experiments of comparision of the different algorithms and find that the FPPM algorithm has the best performance in searching speed within these algorithms. We also test the real data to make sure that the FPPM algorithm has better performance than the originals, by utilizing Snort 2.6.0 with different searching algorithms to trace DARPA 1999 week 2 dataset. The ruleset for Snort 2.6.0 has 2830 rules and the dataset has 1753377 packets. The experimental result shows that the Snort takes 5.87 seconds to process all the packets by the Wu-Manber algorithm, 4.64 seconds by the Aho-Corasick algorithm and 4.06 seconds by the FPPM algorithm. The differences between the simulation results and the real data results may due to the different characteristics of the randomly generated rulesets and the existent rulesets, and the FPPM algorithm has the best processiing performance both in simulation and real test.

Chapter 6

Conclusion and Future Work

In this thesis we first evaluated the performance of two multi-pattern matching algorithms by different types of patterns. We found that the length of the patterns with the same prefixes could substantially affect the searching speed and the memory consumption of these algorithms. We combined the structure of DFA in the Aho-Corasick algorithm and the structures of list and shift table in the Wu-Manber algorithm to a new prefix table to reduce the effects of pattern length. And we modified these two algorithms at the searching stage by the function-parallelism approach to the FPPM algorithm. The experimental results show that the performance of the FPPM algorithm is better than only using any one of the Aho-Corasick or the Wu-Manber algorithm in the average case. In addition, The FPPM algorithm can alleviate the algorithmic attacks that extensively increase the execution time by elaborately arranged payload.

In the next step, we will make effort to that the FPPM algorithm can dynamically reconstruct the DFA and the list structures at the searching stage by the workload of the processors. The reconstructing between these two structures can prevent the malicious attacks and balance the workload of each processor. The frequency of reconstructing the structures is a big issue because the reconstruction will seize a part of computing energy. In addition, for some pattern groups with special characteristics, the FPPM algorithm can parallelize the groups by adapting the same algorithm, for example, all Wu-Manber algorithm or all modified Aho-Corasick algorithm to accelerate the overall searching speed. With the enhancement of FPPM algorithm, the NIDS can have a large capacity of processing packet flow.

References

[1] CERT. [Online]. Available: http://www.cert.org/stats/

[2] Sourcefire, ``Snort.'' [Online]. Available: http://www.snort.org/

[3] L. B. N. Laboratory, ``Bro.'' [Online]. Available: http://www.bro-ids.org/

[4] E. Sommer and M. Strait, ``Application layer packet classifier for linux.'' [Online]. Available: http://l7-filter.sourceforge.net/

[5] S. Antonatos, K. Anagnostakis, and E. Markatos, ``Generating realistic workloads for network intrusion detection systems,'' *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 1, pp. 207--215, 2004.

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

[7] J. Beale and R. Alder, *Snort 2.1 Intrusion Detection*. Syngress, 2004.

[8] S. Antonatos, K. Anagnostakis, E. Markatos, and M. Polychronakis, ``Performance analysis of content matching intrusion detection systems,'' in *Applications and the Internet, 2004. Proceedings. 2004 International Symposium on*, 2004, pp. 208--215.

[9] P. Lin, Y. Lin, Y. Lai, and T. Lee, ``Using string matching for deep packet inspection,'' *Computer*, vol. 41, no. 4, pp. 23--28, 2008.

[10] R. Boyer and J. Moore, ``A fast string searching algorithm,'' 1977.

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

[12] A. Aho and M. Corasick, ``Efficient string matching: an aid to bibliographic search," 1975.

[13] P. Lin, Z. Li, Y. Lin, Y. Lai, and F. Lin, ``Profiling and Accelerating String Matching Algorithms in Three Network Content Security Applications," *IEEE Communications Surveys and Tutorials*, 2006.

[14] M. Norton, ``Optimizing pattern matching for intrusion detection," *white paper, Sourcefire Inc*, 2004.

[15] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, ``Deterministic memory-efficient string matching algorithms for intrusion detection," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, 2004.

[16] B. Haagdorens, T. Vermeiren, and M. Goossens, ``Improving the performance of signature-based network intrusion detection sensors by multi-threading," in *Proc. of 5th Intl. Workshop on Information Security Applications*. Springer, 2004.

[17] R. Stallman, ``gcc." [Online]. Available: http://gcc.gnu.org/

[18] N. Nethercote, ``Valgrind." [Online]. Available: http://valgrind.org/