Journal of Engineering Science and Technology
Vol. 4, No. 2 (2009) 220 - 230

PARALLEL QUICK SEARCH ALGORITHM TO SPEED PACKET PAYLOAD FILTERING IN NIDS

ADNAN A. HNAIF*, MOHAMMAD ALHALAIQAH, OMAR ABOUABDALLA, SURESWARAN RAMADASS, MOHAMMED M. KADHUM

National Advanced IPv6 Center of Excellent (NAv6) School of Computer Science, Universiti Sains Malaysia (USM), 11800 Minden, Penang, Malaysia
*Corresponding Author: adnan@nav6.org

Abstract

An Intrusion Detection System (IDS) is a system to detect intruders who try to hack in to the network and steal information and report them to the network administrator. There are many tools used in this field, snort consider one of the most tools mostly used in Network Intrusion Detection System (NIDS). In spite of consuming 31% of total processing due to string matching, and 80% of total processing in case of web-intensive traffic, snort using its rule sets to determine which packets are allowed to pass and which are rejected. In this paper, we parallelized the quick search algorithm using OpenMP and Pthread (Posix) using C language and made a comparison between them; we determine the required number of threads according to many factors. By doing this, we managed to speed up the filtering process for more than 40% and finally. We applied the proposed method into NIDS to enhance the speed of matching process between incoming packet contents and snort rule sets.

Keywords: NIDS, Exact string matching algorithms, Snort, OpenMP, Pthread

1. Introduction

Experiments prove that on high speed networks, software alone is not enough to process all traffic on high speed link [1]. Many attempts have been made to improve hardware as an alternative to software. In spite of the fact that software has a slow speed, it only performs lightweight processing on low speed network links, compared to hardware, which is faster and performs intensive processing on network traffic and supports much higher network output [2]. As we know, the Internet speed has increased over time, and because of the existence of intruders, who try to sneak away. The system exploits the defects of software in a high-speed link. Therefore,

Abbreviations

CPU	Central Processing Unit.
ICMP	Internet Control Message Protocol.
IDS	Intrusion Detection System.
OTN	Option Tree Node
NIDS	Network Intrusion Detection System.
RTN	Rule Tree Node
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.

there is a need to develop a new technique to prevent these intruders from accessing unauthorized data.

Through analyzing the Snort rule sets, there are over 3100 rules in Snort 2.3.3 [3]. Snort (http://www.snort.org) relies on pattern matching to determine the attackers. The number of rules is growing from time to time; thus, snort divides its rule sets into two dimensional linked lists (see Fig. 1). First: Rule Tree Nodes (RTNs) which hold the main information of each rule such as source/destination address, source/destination port and protocols type (TCP, ICMP, UDP). Second: Option Tree Nodes (OTNs) which hold the information for various options that can be added to each rule such as TCP flags, ICMP codes and types, packet payload size and a major bottleneck for efficiency and packet contents. These two structures are organized into chains, where RTNs are strung from left to right and the OTN's hang down from the RTNs [4].

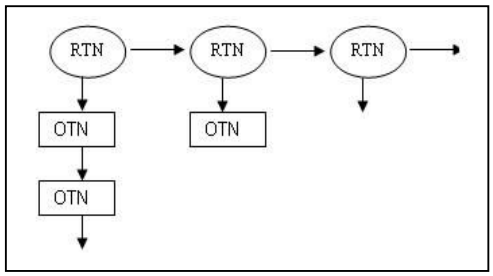

Fig. 1. Structure of Snort Rule Sets [4].

In this paper we parallelized quick search algorithm to enhance NIDS packet payload filtering speed. The rest of this paper is organized as follows: Section 2 introduces some of the existing exact string matching algorithms and previous works in NIDS; Section 3 describes our proposed method and OpenMP and Pthread algorithms; Section 4 reports the results; and concludes the paper in Section 5.

2. Exact String Matching Algorithms and Previous Works in NIDS

2.1. Exact string matching algorithms

2.1.1. Boyer-Moore algorithm

The Boyer-Moore algorithm is considered one of the most famous exact string matching algorithms of many strings against a single keyword, and it is very fast in practice. The algorithm uses two functions to shift the window to the right, the first function called "bad character shift", which means to start working by comparing from right to left beginning with the rightmost one. The second function called "good suffix shift", which means to start working by comparing from right to left too, but in case of mismatching it looks for next occurrence of a substring which has been matched before [4].

2.1.2. AC_BM algorithm (Aho-Corassick, Boyer-Moore)

This algorithm examines the text from right to left and uses a common prefix approach instead of a common suffix approach. The keyword tree moves from the right end of the packet payload to the left while the character comparisons are performed from left to right.

The AC_BM algorithm can be used by one of the two available ways; the first way is "bad character" which is similar to the shift of Boyer-Moore algorithm, which means that if mismatch occurs, then it shifts to the next occurrence in some other keywords in the pattern tree, if there is still mismatch, it will be shifted to the length of the smallest pattern in the tree.

The second use is "good prefix shift" which is similar to "bad character" but here it shifts to the next occurrence according to the smallest pattern length [4].

2.1.3. Quick search algorithm

Quick search algorithm is simplified of Boyer-Moore algorithm, but it uses only the "bad-character shift", also it works like Horspool algorithm by works on one of two shifts of pattern. This algorithm is easy to implement and very fast in practice for short and large patterns (http://www-igm.univ-_ mlv.fr/~lecroq/string/), in spite of quick search algorithm need a quadratic worst case time complexity in searching phase but also it has a good practical behaviour.

2.1.4. Horspool algorithm

Horspool algorithm is almost like quick search algorithm and Boyer-Moore algorithm but in a slightly different way, Horspool algorithm works in any order, and average number of comparisons for one text character is between $1/\sigma$ and $2/(\sigma+1)$ (http://www-igm.univ-mlv.fr/~lecroq/string/).

Table 1 summarizes the complexity of some algorithms for preprocessing phase and searching phase.

Table 1. Complexity of Some Exact String Matching Algorithms [5].

Algorithm	Complexity		
	Preprocessing Phase		Searching Phase
	Space	Time	
Boyer-Moore	O(m + |∑|)	O(m + |∑|)	A: O(mn)
Horspool	O(|∑|)	O(m + |∑|)	A: O(mn)
Quick Search	O(|∑|)	O(m + |∑|)	A: O(mn)

(where A is average case, n: text size, and m: pattern size

2.2. Previous works in NIDS

Li and Yang [6], classified each packet into two fields: Header and Content, and they defined a packet filter as a rule set, since these rule sets determine which packets are allowed to pass and which are reject, when incoming packets are arrived, they inspect its header information, if it matches with one rule sets or not.

In Addition, they divided packets filter into two policies: first policy is to allow all packets to pass except specific types of packets, second policy which is consider contrary of the first policy, is to reject all except specific types of packets. Thus, their algorithm is divided into two parts: first part is in charge of building the decision tree, while the second part is the searching part which is used to find the matching rule in the decision tree for an input packet. Yamashita and Tsuru [7] consider packet filter consist of two sequence procedures. The first procedure is to classify each incoming packet into one of three possibilities: access, drop or forward, according to the rules and content of the packet, the 2nd procedure is the action it will take according to the classification procedure.

Mosqueira-Rey et al [8] classified Intrusion Detection System into two categories: Misuse detection and Anomaly detection. The former depends on signatures which are registered previously in the machine to detect attacks, by meaning, database must be updated periodically to ensure full protection. While the later detects any suspicious activities in the system which are deviate from normal activities based on learning process. They do not use snort rule sets, thus they designed an agent similar to snort but in java environment, they used Rete algorithm for pattern matching process, specifically, implementation in java language was used, and a parser was implemented that convert snort rule sets into Drools rules.

3. Proposed Method

3.1. Multi-core architecture

3.1.1. OpenMP and Pthread programming model

OpenMp can be used to specify shared-memory parallelism, because it is a collection of compiler directive and environment variables. Compiler directive used to control access to code regions updating shared data. Environment variable with library routines used to control execution characteristics of OpenMP code. While Pthread is a posix library which defines an application programming interfaces to the C programming language.

3.1.2. OpenMP and Pthread algorithm

By using OpenMP, operating system distributes input to the number of threads, while in Pthread we can control the distribution of input on threads. Figure 2 represents OpenMP algorithm while Fig. 3 represents Pthread algorithm.

OpenMP Algorithm

Input: Stream of Packets
Output: Match or Mismatch with rule set

```
Step 1: Determine number of threads
Step 2: start time
Step 3 : pragma omp parallel
     {
Step 4: let tid = Number of threads
Step 5: check if (tid == 0)
                        {
N_th =Number of threads
                        }
Step 6: pragma omp for schedule (static,chunk)
Step 7: loop to parallelize input.
Step 8: call PreQsBc function
Step 9: call QS function
    }
Step 10: stop time and calculate elapsed time.
```

Fig. 2. OpenMP Algorithm.

Pthread Algorithm

Input: Stream of Packets
Output: Match or Mismatch with rule set

```
Step 1: Determine number of threads
Step 2: start time
Step 3: calculate traffic size.
Step 4: distribute portion of each thread.
Step 5: create threads and call function which we want to
parallelize
   {
pthread_create(andtid,NULL, thread_function, ptr);
   }
Step 6: Join threads
Step 7: stop time and calculate elapsed time.
```

Fig. 3. Pthread Algorithm.

3.2. System design

Packet capture is a process to capture all packets between sender and receiver in both directions, and then all packets will be sent to the filter process, which checks the incoming packets header and contents, and if it matches with any rule of the rule sets, the packet will be discarded (depends on the rule action).

As shown in Figs. 4 and 5, we separate each incoming packet into two portions (header and content), and classify the rule sets into two linked list, one for header rule sets and one for contents.

Figure 4 shows the architecture of packet header where: SA is the source address, SP is the source port, DA is the Destination address, DP is the destination port , PROT is the protocol and content is the payload of the packet.

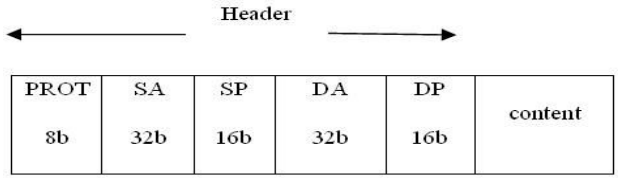

Fig. 4. Some of the Header Fields and Their Widths, and Content.

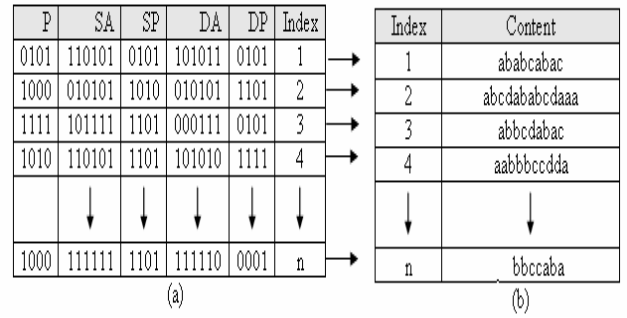

Fig. 5. Design of the Rule Sets Based on (a) Header Rule Sets, (b) Content.

Our proposed method based on studying all the factors to control how many threads required for achieve minimum execution time to process all incoming packets. We used OpenMP and Pthread in our programming using C language and we made a comparison between them. Finally, we summarized our result to show how many threads required to achieve the best result in real traffic size depends on different factors.

3.3. Factors to control the number of required threads

The process of controlling the number of the required threads can be affected by several crucial factors. These factors could be related to the size and the length of the packet content being transmitted, or related to the available hardware on the machine.

In terms of size and length of the packets content there are several factors affects to the number of threads such as: size of traffic, size of rule set, length of packet content and length of rule set. While in terms of the available hardware, the CPU speed, number of CPUs in platform and memory size plays the main factors which affect the number of threads. Knowing all mentioned factors, any change in any one of them will affect the results.

3.4. Working example

Suppose we have a traffic size of 1 mb, and rule set size of 5k, and we want to filter all traffic in shortest time. First, we have to mention how threads are work? Second, what is the relation between factors to determine required threads?

Let us consider that traffic implement in *n* rows of an array, each row exemplify an input (see Fig. 6).

Row	Packets Contents
0	a$vandgsc#\sghgssaaaaaaaaaa
1	b@wwghgsdsdsdh
2	aabbaacc
3	aaaaaaaaaaaaaabbbbbbbbbbbb
⋮	⋮
n	Fffffaaaaafb%4$12*sbb^sb

Fig. 6. Example of Content of Incoming Packets.

Now, we want to process entire array using 1 thread. In this case, thread number 1 will process the entire array from row 0 to row n. but if we change number of threads to 2, then each thread will process its own rows. After each thread finished process present row, it will rush to process the next row and etc… until finishing entire array (see Fig. 7).

Row	Packets Contents	Process by thread #
0	a$vandgsc#\sghgssaaaaaaaaaa	1
1	b@wwghgsdsdsdh	1
2	aabbaacc	2
3	aaaaaaaaaaaaaabbbbbbbbbbbbb	1
4		1
5		2
6		2
7		1
⋮	⋮	⋮
n	Fffffaaaaafb%4$12*sbb^sb	1

Fig. 7. Distribution of 2 Threads into Entire Array.

It is not necessary that each thread will process the same number of inputs because this process depends on the speed of the thread.

Of course, each factor has its own effect to the required time to process incoming packet content, and we will show that on our results.

4. Experimental Results and Discussion

As we have mentioned, each factor has its own effect, for that we implemented our method on the following platform: Master Node: 2x Quad-Core Intel Xeon 1.6GHz, 2x4MB Cache, 1066MHz FSB 8GB (4x2GB), DDR-2 667MHz ECC 2R Fully Buffered Memory 2x 750GB, while Workers Node: 2x Quad-Core Intel Xeon 1.6GHz, 2x4MB Cache, 1066MHz FSB 8GB (4x2GB), DDR-2 667MHz ECC 2R Fully Buffered Memory 250GB. On the other hand we changed the other factors to determine the number of required threads to achieve optimal results.

4.1. Results by using OpenMP programming

Two scenarios are carried out. In each scenario, we changed one of the factors to show effects on the number of threads required to process incoming packet content.

Scenario 1: in this scenario, we fixed rule set size (3k) while size of inputs is variables (40k, 80k, 120k, and 160k) (see Fig. 8a).

Scenario 2: in this scenario, we increase rule set size into 5k and 7k with fixed input size 160k.

As shown in Fig. 8a, the best number of threads required to process input sizes with fixed size of rule set, is 2 threads, the only different case which required 3 threads is when input size = 40k. In scenario 2, the best number of threads is 4 threads in rule set of size 5k and 5 threads in rule set of size 7k. (See Fig. 8b).

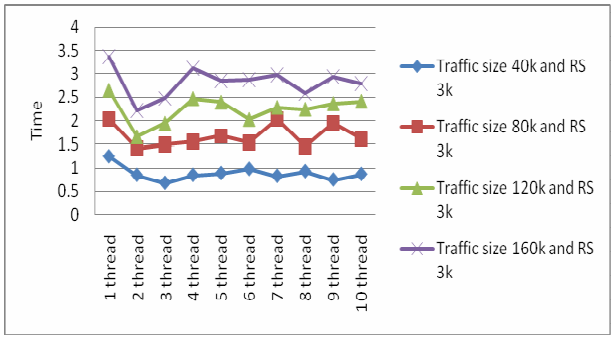

Fig. 8a. Result by Using OpenMP with Different Input Size and Fixed Rule set.

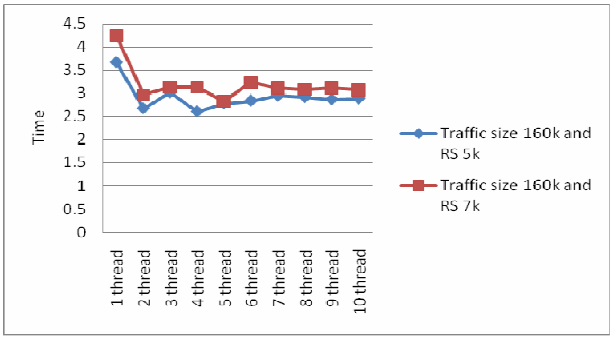

Fig. 8b. Result by Increase Rule Set Size and Fixed Input Size with 160k using OpenMP.

4.2. Results by using Pthread programming

Here, we also carried our experiment in two scenarios. Figures 9a and 9b represent the same scenarios as described in subsection 4.1 but in this case by using Pthread.

As shown in Fig. 9a, the best numbers of required threads when rule set size is fixed is 2 threads, while the best number of required threads when traffic size is fixed is 5 (Fig. 9b).

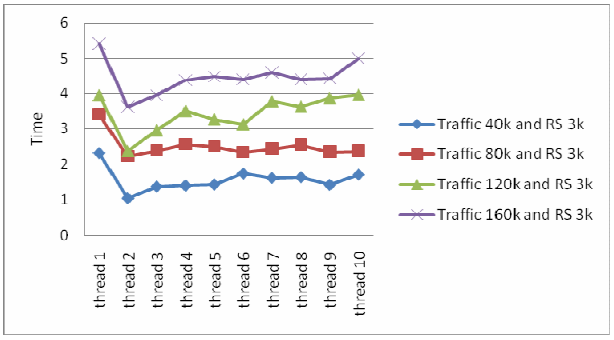

Fig. 9a. Result by Using Pthread with Different Input Size and Fixed Rule Set.

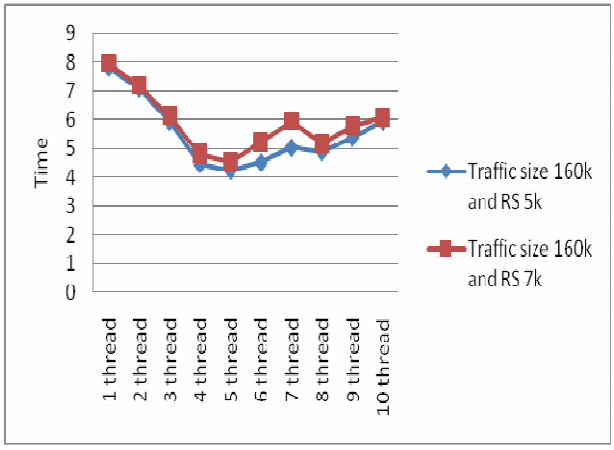

Fig. 9b. Result by Increase Rule Set Size and Fixed Input Size with 160k Using Pthread.

In subsections 4.1 and 4.2, we print the output for each thread which lead to time increase, while in real test (size of inputs are somewhat close to real traffic size), we did not print the output, for that we can see some different in time value from real traffic and unreal traffic (see Fig. 10).

As shown in Fig. 10. The OpenMp and Pthread have been improved until 3 or 4 threads, after that the time which required to achieve traffic size with fixed rule set in somewhat are fixed, thus if we will use more than 10 threads the time will be approximate within the range. To show the strength and direction between

number of threads and time, we calculate the correlation for OpenMp and Pthread. The correlation of OpenMp is (-0.703482427) which indicate that the relation between number of threads and time is strong, and also the correlation for pthread is approximate to (-0.681119075) which also strong.

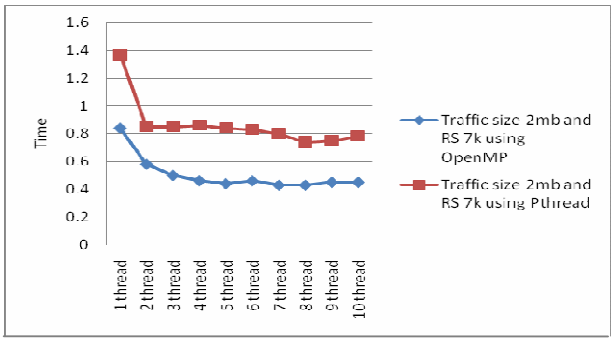

Fig. 10. OpenMP and Pthread in Real Traffic Size.

5. Conclusions

From the results in section 4, we can note that when rule set has a fixed size and size of inputs are variables, the best number of threads required (in the two implementations) are 2 threads, while the number of threads required when rule set size increased to 5k or 7k and fixed size of input 160 k are 5 threads. These results can lead us to the fact that the number of threads required to process size of input depending on size of packet, size of rule set, length of packet content and length of rule set, also depending on CPU speed, number of CPUs and memory size.

In this work we have presented a proposed method to enhance the speed of the packets content filtering. We divided the work into two phases: first, parallelized the Quick Search Algorithm using OpenMp and Pthread. Second we reached to the fact that the numbers of threads required to process size of input are depending on mentioned factors, we speed up the filtering process for more than 40% when we applied the proposed method into real traffic size.

Acknowledgements

We would like to thank the University Sains Malaysia – School of Computer Science - for its support that enabled us to complete this work.

References

1. L. Schaelicke; T. Slabach; B. Moore; and C. Freeland. (2003). Characterizing the performance of network intrusion detection sensors. *In Proceedings of Recent Advances in Intrusion Detection (RAID 2003)*, September 2003.
2. H. Song; T. Sproull; M. Attig; and J. Lockwood. (2005). Snort offloader: A reconfigurable hardware NIDS Filter. *International conference on Field programmable logic and applications*, 493-498.

3. Xinsheng Wang; and Hongxia Li. (2006). Improvement and implementation of network intrusion detection system. *Journal of Communication and Computer*, 3(1), 49-52.
4. C.J. Coit; S. Staniford; and J. Mchlerney. (2001). Towards faster string matching for intrusion detection or exceeding the speed of snort. *In DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, Anaheim, CA, June 2001.
5. A.N.M. Ehtesham Rafiq; M. Watheq El-Kharashib; and Fayez Gebali . (2004). A fast string search algorithm for deep packet classification. *Computer Communications*, 27(12), 1524-1538.
6. Chunyan Li; and Yongtian Yang. (2003). Predictable packet filtering based on decision tree classifies. *IEEE International conference on Robotics, Intelligent Systems and Signal Processing*, 2003*,* Vol. 2, 1345-1349.
7. Yoshiyuki, Yamashita; and Mosato Tsuru. (2007). Code optimization for packet filters. *Proceedings of the 2007 international symposium on Applications and the internet workshops (SAINTW' 07) IEEE*, 86-86.
8. Eduardo Mosqueira-Rey; Amparo Alnaso-Betanzos; Belen Baldonedo del Rio; and Jesus Lago Pineiro. (2007). A misuse detection agent for intrusion detection in a multi-agent architecture. *Lecture notes in Artificial Intelligence*; Vol. 4496, *Proceedings of the 1st KES International Symposium on Agent and Multi-Agent Systems:* Technologies and Applications, 466-475.
9. Ranjit Noronha; and D.K. Panda. (2007). Improving scalability of OpenMP applications on multi-core systems using large page support. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 1-8.