SCALABLE STRING MATCHING FRAMEWORK ENHANCED BY PATTERN CLUSTERING

Bo Xu[1, 3], Kai Zheng[2], Yibo Xue[3, 4], Jun Li[3, 4]
[1]Dept. of Automation, Tsinghua University, Beijing, China
[2]China Research Lab, IBM, Beijing, China
[3]Research Inst. of Info. & Tech., Tsinghua University, Beijing, China
[4]Tsinghua National Lab for Information Sci. & Tech., Beijing, China
xb00@mails.tsinghua.edu.cn

ABSTRACT

String matching plays an important role in content inspection based applications such as network intrusion detection/prevention and anti-virus. It is facing critical performance challenges due to the rapid increase in network bandwidth and the expansion in pattern set size. With multicore processors emerging as the dominant network processing platform, traditional one-dimension workload distribution model via flow-based traffic parallel processing can not fully exploit their computing power and cache hierarchy. In this paper, a scalable string matching framework is proposed by introducing another workload distribution dimension in pattern set. This framework distributes workloads in two dimensions: the network traffic dimension and the pattern set dimension. A novel pattern clustering mechanism named *PCM* is presented to optimize the pattern set partitioning. Experimental results show that the proposed framework obtains a throughput speedup of 60% compared with the traditional one-dimension workload distribution mode on real-life rule sets while the *PCM* pattern clustering mechanism further improves the overall throughput by 15%~20%. The framework can adapt to various string matching algorithms and the *PCM* scheme can be applied to different leap-based algorithms.

Keywords: string matching, clustering, intrusion detection, load balancing

1 INTRODUCTION

As the Internet becomes one of the most critical infrastructures of modern society, network security is attracting more and more concern. Being the most widely deployed security device, firewall controls the information access between internal and external networks by inspecting the packet headers. However, there are numerous types of malicious attacks that deceive firewalls by hiding threaten patterns in packet payloads, such as intrusions, viruses and spam. Consequently, content inspection based devices such as network intrusion detection/prevention systems (NIDS/NIPS), virus scanners, and spam filters emerge to complement the functionalities of firewalls. Besides, unified threat management system (UTM) appears to integrate all the packet filtering applications together. In these content inspection devices, the most challenge task is to accelerate string matching speed to catch up with the booming of network bandwidth as well as the expansion of pattern set sizes.

Traditional string matching architectures based on general purpose processors suffer severely from the limitation in computing power and the lack of parallelism. On the other hand, hardware solutions, which are typically based on FPGA/ASIC, try hard to exploit various levels of parallelism with integrated heterogeneous accelerators on-chip. However, hardware schemes are restricted from publicly deployment due to their intrinsic insufficiencies: first, hardware platforms like FPGA/ASIC have high price and long time-to-market term; second, they are specifically designed chips with poor programmability and portability.

Compared with hardware solutions, open source IDS products, such as Snort [1] and Bro [2], provide much more portable, flexible and economical mechanisms for detecting attacks or intrusions. Meanwhile, multicore processor is emerging as a competent alternative to serve as the platform for today’s network security appliances, owing to its high programmability compared with hardware approaches and high processing power compared with general CPUs. Hence, implementing software schemes on multicore processors becomes an attractive solution.

This paper focuses on improving the searching speed and the scalability of string matching systems on multicore platforms. Conventional solutions on multicore platforms usually dispatch the incoming packets to different processing cores based on flow-

Part of this research is conducted during Bo Xu’s internship in IBM CRL.

level traffic load balancing, which is called the one-dimension workload distribution model in this paper. This one-dimension solution is easy to implement, for the cores share the same pattern set and hence the same data structures. Accordingly, the entire string matching speed can be improved simply by increasing the number of cores, given that the packets flows can be evenly dispatched.

However, the one-dimension solution faces tough performance challenge when the pattern set size grows, since most of existing string matching algorithms scale poorly with large pattern sets. For instance, the prevailing DFA-based AC [3] algorithm consumes linearly increasing memory storage with the increase of pattern set size, which will cause performance decline due to the increase in cache misses. Meanwhile, leap based algorithms such as WM [4] and RSI [5] suffer from the decrease in average leap value caused by the expansion of pattern set size. Take WM for example: with the pattern size growing, the occurrence probabilities of each character pair will grow as well. If the two suffix characters of the pattern set happen to cover all the values from 0x0000 to 0xFFFF, the Bad-Character SHIFT table will get zero in every entry, indicating that no leap could be obtained, which will further result in brute-force comparisons at each position, thus completely conceals the benefit of the leap-based algorithms.

To overcome the disadvantages of the one-dimension workload distribution model, this paper proposes a two-dimension workload distribution framework by introducing an additional dimension on the pattern set. Consequently, an efficient and scalable string matching architecture is constructed by appropriately partitioning the pattern set as well as balancing the incoming traffic. However, this two-dimension framework brings a new issue of how to partition the patterns into subsets to gain algorithmic benefits, since different string matching algorithms perform distinctively on the same pattern set while the same string matching algorithm might performs distinctively on different pattern sets.

In this paper, the proposed two-dimension framework firstly groups the patterns into short and long categories, which are suitable for DFA-based and leap-based algorithms respectively, and then partitions the long patterns into smaller subsets via a novel pattern clustering mechanism, which exploits the intrinsic characteristics of the adopted algorithm. Afterwards, on the second dimension, the framework dispatches the incoming packets via flow-level traffic balancing. The contributions of this paper can be summarized as follows:

- A scalable string matching framework on multicore platform is proposed by introducing a new dimension in pattern set partitioning into workload distribution. Compared with the traditional one-dimension workload distribution model, the new two-dimension framework better exploits the computing power and cache utilities of the multicore platforms, which achieves an overall performance speedup of 60%~80%.
- A novel pattern clustering mechanism called *PCM* is presented to optimize the pattern set partitioning according to the characteristics of the adopted algorithms. Theoretical cost function of the optimization is illustrated and experimental results show that the *PCM* algorithm contributes an additional performance gain of 10%~20% into the two-dimension framework compared with random pattern clustering mechanism.

The rest of this paper is organized as follows. Section Ⅱ analyzes the challenges in string matching on multicore platforms. Section Ⅲ describes the proposed two-dimension workload distribution framework and the *PCM* pattern clustering mechanism. Experimental results and analysis are given in Section Ⅳ. Related works are discussed in Section Ⅴ and finally conclusions are drawn in Section Ⅵ.

2 PROBLEM ANALYSIS

String matching is widely employed in various network security devices including IDS/IPS, virus scanners, and spam filters, etc. In these devices, string patterns are used to denote different kinds of attacks or infections. The emerging of multicore processors facilitates the research on software-based string matching architectures and an intuitive idea is to dispatch the incoming packets into different cores to scan the traffic in parallel. Although this model exploits the parallelism of multicore platforms via flow-level traffic load balancing, its scalability faces tough challenge because each core should deal with the entire pattern set with numerous patterns.

As a well-known open source intrusion detection system, Snort [1] has 5,831 patterns in the rule set of March 2008 and the number is keep growing. It is observed in our tests that the expansion of the pattern set size causes performance decline of string matching algorithms, either DFA-based or leap-based. To interpret this phenomenon, AC is taken as the representative of DFA-based algorithms and MRSI [6] is taken as the representative of leap-based algorithms.

Fig. 1-(a) depicts the searching speed of the AC algorithm, which is tested on one core of the AMD Opteron processor 270 CPU with 64KB L1 cache. It shows that the searching speed of AC keeps decreasing with the increase of pattern number and the performance drops rapidly when the pattern set contains more than 512 patterns. It is believed that the performance decline is due to the increasing cache miss ratios in data accesses, caused by the limited cache size as well as the increasing number of patterns. This affirmation is validated in Fig. 1-(b), which depicts the data cache miss ratios captured by cache profiling and shows that the data cache miss ratios keep increasing with the number of patterns. Therefore,

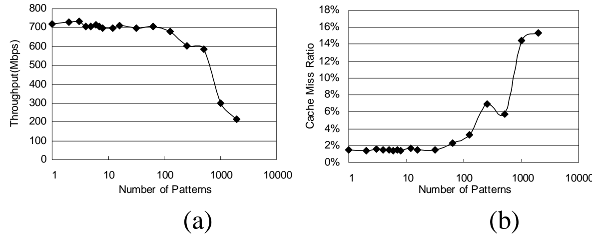

Figure 1: Throughput and cache miss ratio of AC

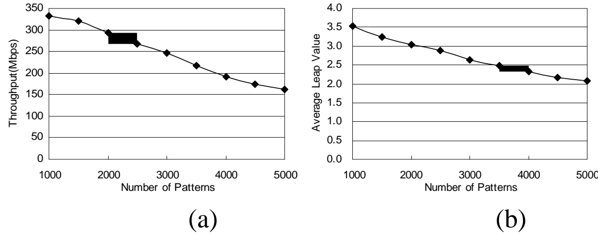

Figure 2: Throughput and average leap value of MRSI

large pattern sets are likely to induce high cache miss ratios, and thus cause low searching speed. For instance the throughput with 1,992 patterns in Fig. 1-(a) is only about one third of the throughput with less than 100 patterns.

Fig. 2-(a) shows the searching speed of MRSI on the same core as used in testing AC. The patterns are selected from Snort patterns with length no shorter than 6 bytes. It is shown that the throughput is decreasing as the number of patterns grows. This should be attributed to the leap-based characteristic of the MRSI algorithm, for the occurrence probability of each character pair at certain position increases with the growing of the pattern number and accordingly the average leap value decreases. Fig. 2-(b) shows the curve of the average leap values with increasing number of patterns, which is consistent with the above deduction. As a conclusion, the expansion in pattern set size will cause performance decline in leap-based algorithms as well. Moreover, leap-based algorithms have intrinsic defections to cover short patterns. For example, WM cannot process patterns shorter than 4 byes and MRSI cannot process patterns shorter than 6 bytes.

On multicore platforms, the traditional one-dimension workload distribution model dispatches the packets into cores at flow-level granularity and employs the same algorithm in all the string matching engines. An overview of the traditional framework is shown in Fig. 3, where the cores share the entire pattern set and the packets belonging to the same flow are processed in the same core. In this model, although the flow-based scheduling scheme can exploit the parallelism of multicore processors, the performance decline caused by the expansion of pattern set size are not solved, which will depress the overall inspecting speed of the framework. A better solution is to partition the pattern set into smaller subsets and adopt suitable algorithms on each subset, which motivates our work in this paper.

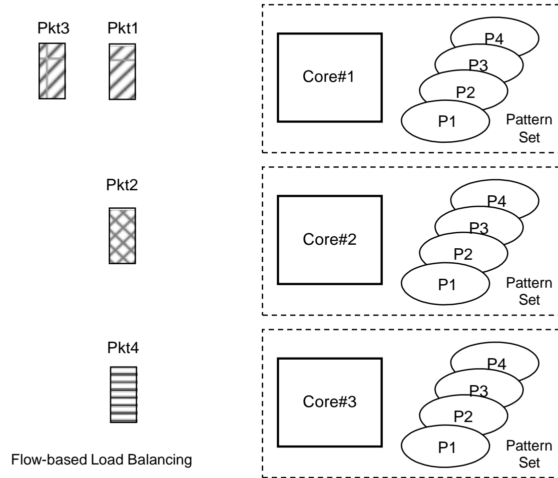

Figure 3: One-dimension workload distribution model

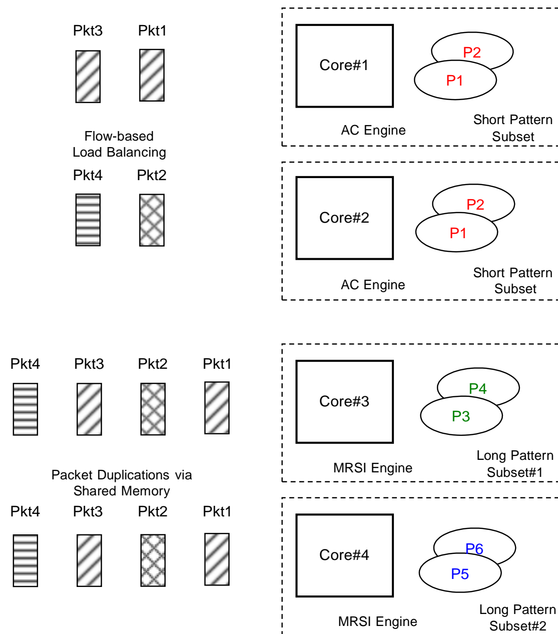

Figure 4: Two-dimension workload distribution framework

3 OUR SOLUTION

3.1 Two-Dimension Workload Distribution Framework

To overcome the shortcomings of the traditional one-dimension workload distribution model, we extend it to a two-dimension framework by introducing another workload distribution dimension on pattern set. As a result, large pattern sets can be split into small subsets to avoid the performance decline caused by the extension of pattern sets, and meanwhile different types of string matching algorithms can be selected to adapt the characteristics of each pattern subset.

The novel architecture proposed in this paper is named as two-dimension workload distribution framework. It distributes workload in one dimension

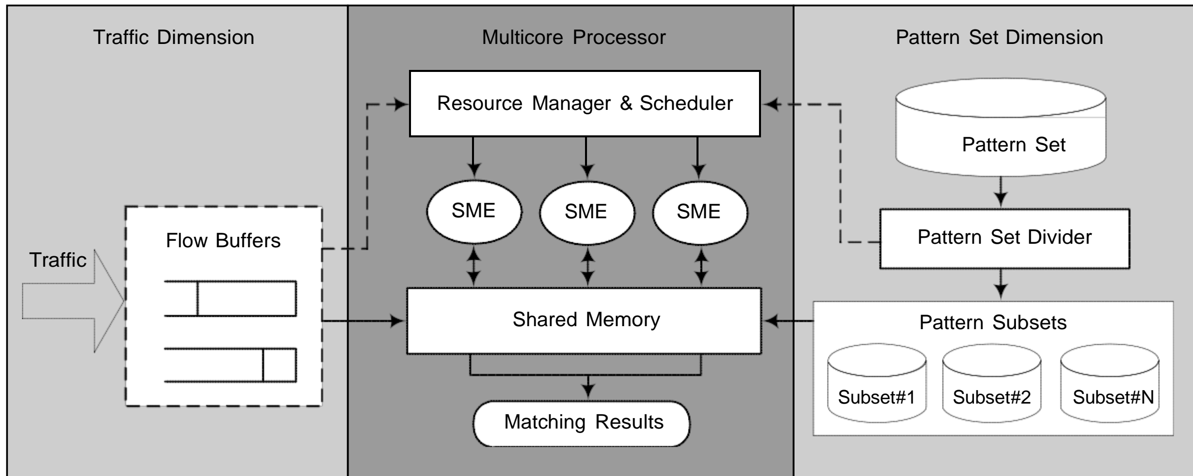

Figure 5: System architecture of the two-dimension workload distribution framework

by pattern set partitioning to exploit the best pattern scattering in the pattern subsets, and then engages flow-based load balancing in another dimension to achieve high systemic throughput.

Fig. 4 illustrates a prototype of the two-dimension workload distribution framework. In the first dimension, the entire pattern set is divided into two categories, the short patterns *P1~P2* and the long patterns *P3~P6,* and further, the long patterns are divided into two subsets: *P3* & *P4* to be processed in Core#3 and *P5* & *P6* to be processed in Core#4. DFA-based and leap-based algorithms are employed to process the short and long pattern subsets separately, where AC and MRSI are selected in this example. In the second dimension, flow-based load balancing is employed to distribute the incoming packets evenly to different cores. The packets need to be duplicated to be dispatched into the pattern subsets simultaneously. However, compared with the searching speed of the string matching engines, it is believed that the duplicating and dispatching of the packets will not become the bottleneck of the whole framework.

3.2 System Architecture

As shown in Fig. 5, the proposed framework mainly contains four components: a *Flow Buffer*, a *Resource Manager & Scheduler (RMS)*, *a Pattern Set Divider (PSD)*, and several *String Matching Engines (SMEs).*

Flow Buffer is in charge of storing and reassembling the incoming packets, so that the packets belonging to the same flow are stored sequentially in the same queue. *PSD* is responsible for pattern set partitioning and data structure optimization. It determines how to divide the patterns into subsets in the preprocessing stage, which will be further discussed in Section Ⅲ.C. *SMEs* denote the cores where string matching operations are performed. *RMS* is responsible for allocating the computing resources (cores) in the preprocessing stage according to the expected overall performance. *RMS* also schedules the packets distribution in the searching stage. Detailed explanation of *RMS* is stated in Section Ⅲ.E.

3.3 Pattern Set Partitioning

In the preprocessing stage, *PSD* is in charge of dividing the patterns into subsets for distributed pattern matching in the *SMEs*. There are three steps for pattern set partitioning.

In the first step, string matching algorithms are chosen for the *SMEs*. The selection of algorithms is related to the characteristics of the pattern set. Take Snort rule for example: Fig. 6 shows the length distribution of Snort rules in March 2008, which contain 5,831 patterns. The rule set has 68 patterns with 1 byte, and 1,421 patterns shorter than 6 bytes. Since leap-based algorithms cannot handle short patterns, DFA-based algorithms are required. Meanwhile, since DFA-based algorithms suffer from cache misses with large pattern sets, leap-based algorithms are required to complement DFA-based algorithms. In this paper, AC is selected as the representative of DFA-based algorithms to handle the short patterns less than 6 bytes, while MRSI is selected as the representative of leap-based algorithms to handle the rest long patterns. It should be noticed that the two-dimension framework is independent of the algorithms. AC and MRSI are simply selected as an example to illustrate the advantage of the two-dimension framework.

In the second step, the pattern subset sizes should be decided for the AC and MRSI engines. As known, both of the algorithms suffer performance decline due to the expansion of pattern set sizes. For AC, as illustrated in Fig. 1-(a), it typically has an inflexion in performance due to the increase in data cache miss ratios. Furthermore, the inflexion is mainly determined by the L1 cache size of the CPU. Hence, the proper pattern subset size for AC is determined by the hardware architecture of the multicore platforms. When deployed on the platform in the case of Fig. 1, the proper size should be less than 512 patterns, for the

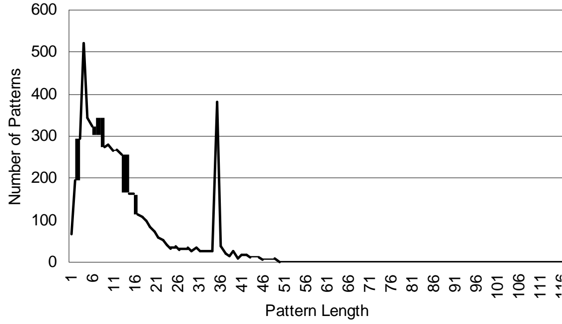

Figure 6: Pattern length distribution of Snort rules

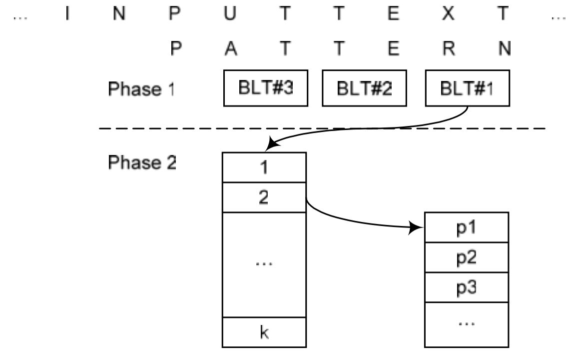

Figure 7: Data structure of MRSI

performance decline accelerates at this point, which makes it an inflexion. The pattern subset sizes for MRSI engines should be determined by the performance expectation on single core. For example, as shown in Fig. 2-(a), if the single core matching speed is expected to be larger than 300Mbps, the pattern subset size should be less than 2,000 patterns.

In the third step, the patterns are classified into different subsets. For DFA-based algorithms like AC, the performance of each subset is primarily determined by the number of patterns rather than the scattering of the patterns in different subsets. However, pattern scattering can significantly influence the performance of leap-based algorithms like MRSI since the leaps are determined by the distribution of the characters in patterns. Consequently, pattern clustering for leap-based algorithms becomes an open issue, which motivates the design of the pattern clustering mechanism in this paper.

3.4 Pattern Clustering Mechanism (PCM)

Pattern scattering in the pattern subsets is of great significance to the searching speed of leap-based algorithms. If the patterns in each subset are treated as a cluster, the pattern scattering in the subsets can be seen as a pattern clustering problem. Mathematically, pattern clustering is an optimization problem of seeking the optimal mapping function $f: P \rightarrow S$ to minimize the sum of cost values $\sum \lambda_j (1 \le j \le m)$, where P is the pattern set $P = \{p_i \mid 1 \le i \le n\}$ and S stands for the m subsets $S = \{S_j \mid 1 \le j \le m\}$.

In this paper, a novel pattern clustering mechanism named the *PCM* scheme is devised to optimize the pattern partitioning for leap-based algorithms. Herein, MSRI is taken as the representative of leap-based algorithms to illustrate the details of the *PCM* scheme, which can easily adapt to other leap-based algorithms with slight modification in the cost functions. The procedure of the *PCM* scheme for MRSI includes:

1) Given the number of clusters *m*, randomly select *m* patterns and place each of them into one subset. A *cost value* is calculated for each subset according to a prescribed *cost function*. So m cost values are gotten as $\lambda_j (1 \le j \le m)$.

2) For each p_i in the other $n - m$ patterns, calculate sum of cost values $\sum_k \lambda_j$, which denotes the total cost with p_i joining the subset S_k. If $\sum_t \lambda_j = \min_{1 \le k \le m} \sum_k \lambda_j$, then put p_i into the subset S_t. Thus, the initial subset-ids of all the patterns are set and the subsets get their initial cost values $\lambda_j^0 (1 \le j \le m)$.

3) Start pattern clustering iterations. In the first iteration, for each p_i, assuming its subset-id is t, change its subset-id from 1 to m and calculate the new sum of cost values $\sum_k^0 \lambda_j$, which denotes the total cost with p_i joining the subset S_k. If $\sum_v^0 \lambda_j = \min_{1 \le k \le m} \sum_k^0 \lambda_j$, change the subset-id of p_i from t *to* v. After all patterns finish the first iteration, some patterns are set new subset-id and the subsets get new cost values $\lambda_j^1 (1 \le j \le m)$.

4) Proceed the pattern clustering iretations until the two consecutive total cost tolerance is smaller than a given positive value ε. Thus, if $(\sum_k^r \lambda_j - \sum_k^{r-1} \lambda_j) / \sum_k^r \lambda_j < \varepsilon$, the iterations will be stopped at iteration r and the patterns get their final subset-ids.

The design of *cost function* has crucial influence on the practical performance of MRSI engines, for it determines how the patterns are clustered. Fig. 7 shows the data structure of MRSI [6], in which the three *Block Leap Tables (BLTs)* stores the longest leaps that can be generated according to the three two-character blocks and the *Potential Match Table (PMT)* stores the possible matching patterns when the leap value in *BLT#1* equals to zero. The last two characters are used as hash indexes in *PMT*. The matching process of MRSI is:

a) Use the three two-character blocks of the input text to index the *BLTs* and get three leap values;

b) Denote the maximum of the three leap values as $L_{\max}$. If $L_{\max} > 0$, then shift the input text by $L_{\max}$;

c) If $L_{\max} = 0$, use the last two-character block of the input text as the index to search *PMT*, and match

the potential patterns one by one to find out the exact matches.

According to the searching procedure of MRSI, it is observed that the leap values in the three *BLTs* and the lengths of the hash lists in *PMT* are the key factors that impact string matching speed. Obviously, MRSI performs better if the leap values are larger or the hash list lengths are shorter. Thus, a good pattern clustering mechanism should aim at obtaining the longest average leap value of the entries in the three *BLTs* and obtaining the shortest average length of the hash lists in *PMT*.

The leap values in the three *BLTs* can be denoted as $L_i^k (0 \le i \le 65535, 1 \le k \le 3)$, and so the average leap value will be $\bar{L} = mean(L_i^k)$. The hash list lengths can be designated as $H_i (0 \le i \le 65535)$, and so the average hash list length will be $\bar{H} = mean(H_i)$. It should be noticed that, if there is a potential match, MRSI needs averagely $\bar{H}/2$ times of memory accesses (naïve comparisons) to find out the exact matches. In contrast, if there is no potential matches, MRSI can shift input text by $\bar{L}$, which can be expressed as $1/\bar{L}$ times of memory accesses in cost. Therefore, if using P_{match} to indicate the average possibility of searching *PMT*, the *cost function* of MRSI can be denoted as Equation 1.

$$\lambda = (1 - P_{match})/\bar{L} + P_{match} * \bar{H}/2 \quad (1)$$

Based on this cost function, the patterns can be clustered into several subsets to optimize MRSI's average performance in the subsets. It has been proved that the *PCM* scheme converges at a locally optimal solution. The proof is omitted here due to the page limit. Since the *PCM* scheme uses a hill-climbing strategy, the convergence point might not be the global optimal solution. However, the locally optimal solution is quite satisfactory according to our evaluation results. The experimental result of the *PCM* scheme's convergence speed is provided in Section Ⅳ.D, which shows the convergence speed of *PCM* is pretty fast. The selection of P_{match} is a tiny issue since it is unknown before the input text is scanned entirely. In practice, it can be estimated according to the prior searching results of the context, though it is not a precise value. The impact of P_{match} on clustering effect is discussed in Section Ⅳ.E.

3.5 Resource Management and Scheduling

RMS is responsible for managing the computing resources and scheduling the incoming traffic. Resource management is to allocate the computing resources (cores) to the string matching engines (*SMEs)* in the preprocessing stage while traffic scheduling is to distribute the incoming traffic to the *SMEs* in the

the overall performance requirement. The aim is that the scanning speed on the subsets should be balanced to ensure high utilization of hardware resources. Thus, given an overall expected throughput, assign enough cores to each subset. Otherwise, if the number of cores is not enough, the cores should be allocated proportionally based on the searching speed on each subset to achieve a possibly maximum throughput, assuming the single core searching speed on each subset is prior knowledge with specific string matching algorithms. A simple resource management algorithm is proposed here:

Assume *PSD* has divided the patterns into *m* subsets $S_1, S_2, \ldots, S_m$, and the single core searching throughputs on the subsets are denoted as $T_1, T_2, \ldots, T_m$. Given the expected throughput *E* and the number of cores *C*, the number of cores needed for subset $S_k (1 \le k \le m)$ should be $\lceil E/T_k \rceil (1 \le k \le m)$. Denote $E_{sum} = \sum_{k=1}^{m} \lceil E/T_k \rceil$. If $C \ge E_{sum}$, the number of cores assigned to subset $S_k (1 \le k \le m)$ is $\lceil E/T_k \rceil (1 \le k \le m)$. Else $C < E_{sum}$, the number of cores assigned to subset $S_k (1 \le k \le m)$ is $\lfloor C \times \lceil E/T_k \rceil / E_{sum} \rfloor, (1 \le k \le m)$. Considering the number of cores should be integers, there might be some cores left. If there are *L* cores left, assign the left cores to the *L* subsets with the *L* largest residues. Residue of subset $S_k (1 \le k \le m)$ is defined

searching stage.

In the preprocessing stage, *RMS* determines how

as $R_k = C \times \lceil E/T_k \rceil / E_{sum} - \lfloor C \times \lceil E/T_k \rceil / E_{sum} \rfloor$, $(1 \le k \le m)$.

In the searching stage, *RMS* acts as a scheduler and decides which core the incoming packets should be distributed. The scheduling has two compulsory steps: first, if the pattern set is divided into *m* subsets, *RMS* duplicates *m* copies of every packet and sends each copy of it to the cores combined with each pattern subset; second, if one pattern subset is assigned with more than one core, *RMS* is responsible for balancing the workload on the cores to approach high CPU and cache utilities. In addition to the second step, *RMS* is required to dispatch the packets at flow level to detect avoidance attacks which span packets.

To summarize, in the two-dimension workload distribution framework, *PSD* takes charge of the first dimension workload distribution on pattern set partitioning while *RMS* takes charge of the second dimension workload distribution on traffic load scheduling.

4 EXPERIMENTAL RESULTS

4.1 Experiment Setup

The proposed two-dimension workload distribution framework and the *PCM* scheme were evaluated on an AMD Opteron multicore platform. The platform has 4 cores running at 2GHz with totally 2GB DDR2 memory. Each core has 64KB L1 cache and supports 4 zero-overhead-content-switched hardware threads. In the tests, AC is selected as the representative of DFA-based algorithms, and MRSI is

Two real-life pattern sets are adopted to evaluate the performance of the two-dimension framework compared with the traditional one-dimension model. One is the Snort rule set in March 2008, including totally 5,831 patterns. Along with the Snort rule set, a real-life trace provided by DEFCON [7] is used as the testing trace, which contains totally 353,799,850 bytes. Another real-life rule set is the rule set of the well-known anti-virus software ClamAV [8]. The ClamAV rule set is larger than 100K patterns, but only 8,000 patterns are randomly selected from it to test the two frameworks, since the AC algorithm cannot handle more than 8,000 patterns on the AMD Opteron multicore platform due to the huge memory occupation and thus the traditional one-dimension model cannot get its throughput. A randomly generated trace is used as the test trace along with the ClamAV rule set.

4.2 Evalution of the Two-dimension Framework

To evaluate the efficiency of the two-dimension workload distribution framework, the overall throughput of the proposed two-dimension framework is compared against the traditional one-dimension model.

In the traditional one-dimension workload distribution model, the entire pattern set is applied in all the string matching engines *(SMEs)*. Since leap-based algorithms cannot handle short patterns, AC becomes the only choice for the string matching algorithm in the *SMEs*. As shown in Fig. 3, each *SME* handle the whole pattern set with AC algorithm. When tested on the AMD Opteron multicore platform, each of the four cores runs a *SME* with AC algorithm, while the incoming traffic is distributed to the four cores with flow-level load balancing. The overall throughput is the average value of three times of tests with each pattern set and its corresponding trace.

When testing the two-dimension workload distribution framework, short patterns with less than 6 bytes are handled by AC while the long patterns are handled by MRSI. As shown in Fig. 4, two cores of the AMD Opteron multicore platform are selected as AC string matching engines, and flow-level load balancing is applied. The long patterns are split into two subsets either with random clustering mechanism *(RCM)* or with the *PCM* scheme, and the other two cores run MRSI string matching engines with the two subsets separately. Every incoming packet is duplicated and sent into the two MRSI engines to scan simultaneously. The overall throughput is a total value of the two AC engines and the two MRSI engines, and an average value of three times of tests with each pattern set and its corresponding trace is taken as the final result.

Table 1 shows the performance of the one-dimension model and the two-dimension framework with the two real-life pattern sets and their corresponding traces on the AMD Opteron multicore platform. With the DEFCON trace and the Snort pattern set which contains 5831 patterns, the one-dimension model gets a throughput of 192Mbps. Meanwhile, the two-dimension framework with random clustering mechanism *(RCM)* gets a throughput of 316Mbps, which is 64% higher than the performance of the one-dimension model. Besides, when the *PCM* scheme is adopted, the two-dimension framework achieves a throughput of 377Mbps, which introduces an extra 19% speedup compared with the *RCM* scheme.

With the ClamAV pattern set which contains 8000 patterns and a randomly generated trace, the one-dimension model gets a throughput of 142Mbps. Meanwhile, the two-dimension framework with random clustering mechanism *(RCM)* gets a throughput of 234Mbps, which is 64% higher than the performance of the one-dimension model. When the *PCM* scheme is engaged, the two-dimension framework obtains a throughput of 274Mbps, which introduces an extra 17% performance gain compared with the *RCM* scheme.

Table 1: Performance of different frameworks

Pattern Set	One-dim	Two-dim	Two-dim (*PCM*)
Snort	192 Mbps	316 Mbps	377 Mbps
ClamAV	142 Mbps	234 Mbps	274 Mbps

Besides, the throughput with the ClamAV rule set is slightly lower than that with the Snort rule set. This is due to the larger number of patterns and longer average pattern lengths in the ClamAV rule set, which will cause performance decline in AC string matching engines.

4.3 Evalution of the PCM Scheme

The *PCM* scheme is the kernel algorithm in *Pattern Set Divider (PSD)*. It determines the scattering of long patterns in the MRSI subsets, which can significantly influence the overall performance of the two-dimension framework.

To evaluate the performance of the *PCM* scheme, experiments were conducted on the real-life DEFCON trace along with the long patterns of the Snort rule set, which includes totally 4,410 patterns with no less than 6 bytes. Since there is no any prior pattern clustering mechanism in the literature, performance comparison is done between the *PCM* scheme and the random clustering mechanism *(RCM)*, which randomly partitions the patterns into the subsets and ensures every subset having the same number of patterns.

For better comparing the performance of the *PCM* scheme and the *RCM* scheme, the long pattern rule set is divided into 2, 4, 8, and 16 subsets to test the overall MRSI string matching performance on the AMD Opteron multicore platform. Furthermore, since the performance of MRSI is determined by the total memory access times, the average leap values, the

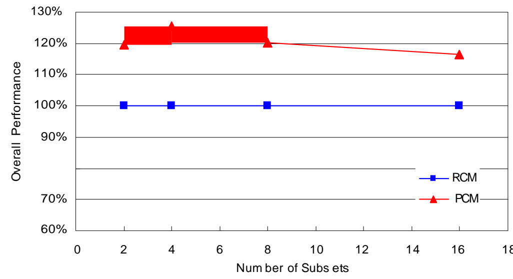

Figure 8: MRSI throughput speedup

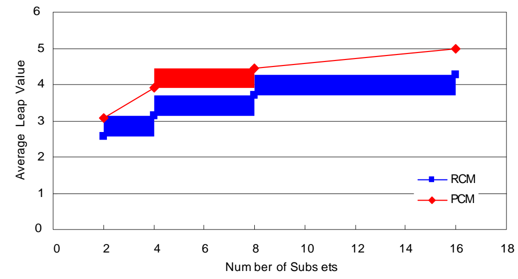

Figure 9: MRSI average leap values

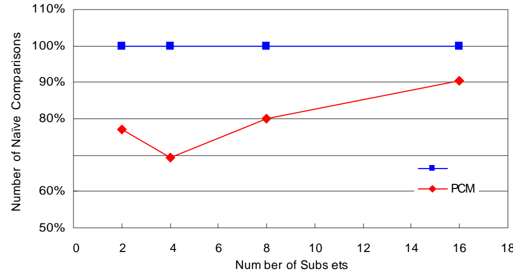

Figure 10: MRSI naïve comparison percentage

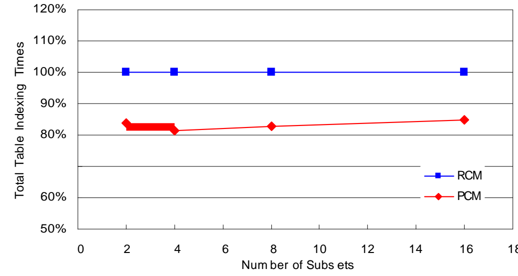

Figure 11: MRSI table indexing percentage

number of naïve comparisons, and the number of table indexing were observed on the two clustering schemes.

Fig. 8 illustrates the MRSI performance gain with the *PCM* scheme compared with the *RCM* scheme. It shows that the *PCM* scheme outperforms *RCM* by 15%~25% when the number of subsets varies from 2 to 16, where *PCM* achieves its highest improvement percentage when the number of pattern subsets equals to 4. One reason is that the AMD Opteron multicore platform has 4 cores, which will get highest CPU utilization efficiency when the number of tasks equals to 4.

Fig. 9 depicts the average leap values of the MRSI algorithm when the patterns are partitioned by the two pattern clustering mechanisms. With a given number of subsets, the overall average leap value is the mean of the average leap values on each subset. It is shown that the average leap value with *PCM* is visibly larger than with *RCM* when the number of subsets varies from 2 to 16. Besides, when the number of subsets increases, the average leap values with the two schemes keep increasing, since the number of patterns in each subset is reduced.

Fig. 10 shows the percentage of naïve comparisons needed by the MRSI algorithm with the *PCM* scheme compared to the *RCM* scheme. With a given number of subsets, the mean of naïve comparisons on each subset is taken as the overall number of naïve comparisons. It is illustrated that *PCM* reduces the naïve comparisons by 10%~30% compared with *RCM* and the reducing percentage reaches its peak value when the number of subsets equals to 4, which partially contributes to the highest performance improvement shown in Fig. 8.

Fig. 11 shows the percentage of table indexing times needed by the MRSI algorithm with the *PCM* scheme compared to the *RCM* scheme. The table indexing means the memory accesses in the *BLTs* of MRSI's data structures, which is part of the overhead in string matching. It is shown that *PCM* reduces the times of table indexing by 15%~25% compared with *RCM*. Besides, the reducing percentage is relatively stable in spite of different number of subsets.

4.4 Convergence Speed of PCM

Table 2 provides the times of iterations needed for the *PCM* scheme with different number of subsets. It is shown that the *PCM* scheme converges quite fast and the required times of iterations increase slowly when the number of subsets grows.

Table 2: PCM convergence speed

#Subsets	2	4	8	16
#Iterations	0	2	3	5

4.5 Selection of Matching Rate

The cost function for *PCM* is $\lambda = (1 - P_{match}) / \overline{L} + P_{match} * \overline{H} / 2$, where the matching rate is a key factor that could impact the clustering effect of the patterns. The matching rate is defined as the times needed for naïve comparisons divided by the total length of the input text. The challenge is that the exact matching rate is unknown before the traffic is inspected against the patterns completely. In implementation, the matching rate can be predicted

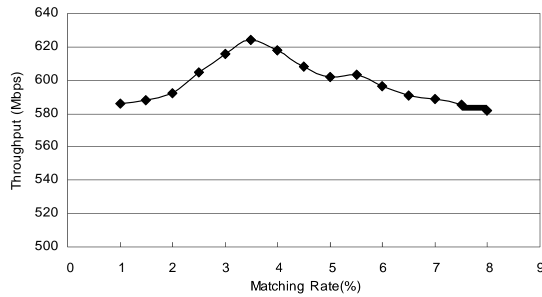

Figure 12: MRSI performance with different matching rates in *PCM*

in a period of time, one hour for example. Otherwise, the matching rate can be set a default value for expedience.

Fig. 12 gives the overall performance of MRSI with different matching rate applied in the *PCM* cost function. In this test, the number of MRSI subsets is set 2 and the experiments are conducted on the AMD Opteron multicore platform. Each subset is assigned with two cores and flow-level load balancing is engaged to distribute the incoming packets. It is shown that the overall performance varies between 580Mbps and 625Mbps and the throughput reaches its peak value when the matching rate equals to 3.5%. In fact, the total matching time is 12,633,581 and the precise matching rate is 3.57%, given the trace of 353,799,850 bytes in length. The results demonstrate that the *PCM* clustering scheme achieves the highest performance when the matching rate approaches the actual rate in practice.

5 RELATED WORKS

Many string matching algorithms and architectures have been proposed in recent years for content inspection. In summary, the previous works can be classified into two categories: one is hardware solutions based on FPGA/ASIC; the other is software algorithms based on general purpose processors. Furthermore, the software algorithms can be classified into DFA-based algorithms and leap-based algorithms.

Aho-Corasick (AC) [3] is the most popular DFA-based software algorithm for multiple pattern string matching. It employs a deterministic finite automaton (DFA) to organize the patterns and reveals matching results in one pass. The advantage of AC is that it only needs search time on the order of $O(n)$, regardless of the number of patterns, where n is the length of the input text. However, AC suffers from two intrinsic deficiencies. First, it does not exploit the heuristics in the pattern set to generate a leap for avoiding unnecessary comparisons. Second, the memory occupation of AC increases linearly with the growing of the pattern set size, which will cause higher cache miss rates and correspondingly lower searching speeds.

Wu-Manber (WM) [4] is the representative of leap-based algorithms that evade unnecessary heuristics. WM inherits the bad character heuristic of Boyer-Moore (BM) [9] algorithm and uses two or three suffix characters to generate a leap. If the leap value does not equal to zero, the matching window can be shifted by this value; otherwise naïve comparisons will be adopted to verify the potential matching patterns indicated by the prefix hash table. WM consumes much less memory than AC, but it needs time-consuming comparisons when no leap could be generated. Moreover, when the pattern set grows large, the probability of getting leaps and the average leap value are inevitable to become smaller, which will deeply impact the string matching speed.

There are many other software algorithms proposed in recent years. Leap-based algorithms including AC_BM [10], Setwise Boyer-Moore [11], E2xB [12], RSI [5], MRSI [6] are presented to leverage the characteristics in pattern sets for improving scanning performance. However, these leap-based algorithms suffer from the same problem with WM that the prospective leap value will become smaller when the number of patterns increases.

Hardware algorithms mainly rely on the specially designed architectures to accelerate string matching. FPGA-based algorithms [13-20] and ASIC-based algorithms [21-27] exploit the parallelism in circuits as well as the abundance of data channels to achieve gigabit-level throughput. However, the high cost, long developing term, and poor flexibility encumber their feasibility of wide deployment.

In recent years, multicore processors have emerged as a competitive candidate for high performance string matching architectures. They are superior to general purpose CPUs for their powerful computing capability and rich memory banks. Besides, multicore platforms surpass hardware circuits by their high programmability, flexibility and portability. This paper aims at exploring a scalable string matching framework on multicore platforms for intrusion detection and deep inspection systems.

6 CONCLUSIONS

In this paper, a novel parallel processing framework, called the *two-dimension workload distribution framework*, is proposed to optimize the multi-pattern string matching on modern multicore platforms. The innovations are motivated by the poor scalability and performance of the traditional one-dimension model, which dispatches the string matching workload only by flow-level traffic load balancing. The proposed two-dimension workload distribution framework improves the traditional model by introducing a new dimension on pattern set partitioning to balance the workload distribution by pattern clustering. It firstly partitions the patterns into subsets according to the selected algorithms' characteristics in the first dimension, and then dispatches the incoming packets via flow-level load

novel pattern clustering mechanism named *PCM* is presented to optimize the performance on the long pattern subsets in MRSI engines.

The *PCM* scheme is devised based on the intrinsic characteristics of the MRSI algorithm, and it can easily be extended to support other leap-based algorithms with slight modifications in the cost function. In addition, the two-dimension workload distribution framework can adapt to various string matching algorithms besides AC and MRSI. The framework provides a platform to split the pattern set into subsets according to the characteristics of different algorithms. It is believed that this scalable framework can support large pattern sets with more than 100K patterns, and the overall searching performance can meet gigabyte level line speed requirement with modest number of cores.

Experimental results show that the proposed two-dimension workload distribution framework achieves a performance speedup of 60% with real-life rule sets, compared with the traditional one-dimension model. Besides, assisted by the *PCM* scheme, the two-dimension workload distribution framework obtains an additional 15%~20% performance gain compared with the random clustering mechanism *(RCM)*. Besides, *PCM* is proved to converge at a locally optimal solution and has a rapid convergence speed, which facilitates its employment in practice.

Future work will be done on designing elaborate resource manager and scheduler to automatically allocate the computing resources on the pattern subsets, and to automatically balancing the overhead in the string matching engines. The final objective is to devise an intelligent architecture that can automatically distribute workload on both pattern and traffic dimensions to form up a well-performed parallel processing system.

7 REFERENCES

[1] M. Roesh: Snort – Lightweight Intrusion Detection for Network, Proc. of the 13th Systems Administration Conference (1999).

[2] Bro intrusion detection system, http://www.bro-ids.org/.

[3] A. Aho and M. Corasick: Fast Pattern Matching: An Aid to Bibliographic Search, Commun. ACM, vol. 18, no. 6, pp. 333-340 (1975).

[4] S. Wu and U. Manber: A Fast Algorithm for Multi-pattern Searching, Technical Report TR-94-17, Dept. Computer Science, University of Arizona (1994).

[5] B. Xu, X. Zhou, and J. Li: Recursive Shift Indexing: A Fast Multi-Pattern String Matching Algorithm, Proc. of the 4th International Conference on Applied Cryptography and Network Security (2006).

[6] X. Zhou, B. Xu, Y. X. Qi, and J. Li: MRSI: A Fast Pattern Matching Algorithm for Anti-virus Applications, Proc. of the 7th International Conference on Networking (2008).

[7] MIT DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[8] ClamAV Anti-virus, http://www.clamav.net/.

[9] R. Boyer and J. Moore: A Fast String Searching Algorithm, Commun. ACM, vol. 20, no. 10, pp. 762-772 (1977).

[10] C. J. Coit, S. Staniford, and J. McAlerney: Towards Faster Pattern Matching for Intrusion Detection, or Exceeding the Speed of Snort, Proc. of the 2nd DARPA Information Survivability Conference and Exposition (2002).

[11] M. Fisk and G. Varghese: Fast Content-based Packet Handling for Intrusion Detection, UCSD Technical Report CS2001-0670 (2001).

[12] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis: E2xB: A Domain-pecific String Matching Algorithm for Intrusion Detection, Proc. of the 18th IFIP International Information Security Conference (2003).

[13] B. L. Hutchings, R. Franklin, and D. Carver: Assisting Network Intrusion Detection with Reconfigurable Hardware, IEEE Symposium on Field-Programmable Custom Computing Machines (2002).

[14] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos: Implementation of A Content-scanning Module for an Internet Firewall, IEEE Symposium on Field-Programmable Custom Computing Machines (2003).

[15] C. R. Clark and D. E. Schimmel: Scalable Pattern Matching for High Speed Networks, IEEE Symposium on Field-Programmable Custom Computing Machines (2004).

[16] Z. K. Baker and V. K. Prasanna: Time and Area Efficient Pattern Matching on FPGAs, IEEE Symposium on Field-Programmable Custom Computing Machines (2004).

[17] Z. K. Baker and V. K .Prasanna: Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs, IEEE Symposium on Field-Programmable Custom Computing Machines (2004).

[18] L. Bu and J. A. Chandy: FPGA Based Network Intrusion Detection Using Content Addressable Memories, Proc. of IEEE International Conference on Field-Programmable Technology (2004).

[19] I. Sourdis and D. Pnevmatikatos: Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching, IEEE Symposium on Field-Programmable Custom Computing Machines (2004).

[20] Z. K. Baker and V. K. Prasanna: High-throughput Linked-pattern Matching for Intrusion Detection Systems, Proc. of Symposium on Architecture for Networking and Communications Systems (2005).

[21] F. Yu, R. H. Katz, and T. V. Lakshman: Gigabit Rate Packet Pattern-matching Using TCAM, Proc. of the 12th IEEE International Conference on Network Protocols (2004).

[22] J. Lockwood: Deep Packet Inspection Using Parallel Bloom Filters, IEEE Micro, vol. 24, pp. 52-61 (2004).

[23] N. Tuck, T. Sherwood, B. Calder, and G. Varghese: Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection, Proc. of INFOCOM (2004).

[24] L. Tan and T. Sherwood: A High Throughput String Matching Architecture for Intrusion Detection and Prevention, Proc. of International Symposium on Computer Architecture (2005).

[25] J. V. Lunteren: High-performance Pattern-matching Engine for Intrusion Detection, Proc. of IEEE INFOCOM (2006).

[26] S. Dharmapurikar and J. Lockwood: Fast and Scalable Mattern Matching for Network Intrusion Detection Systems, IEEE Journal on Selected Areas in Communications, vol. 24, pp. 1781-1792 (2006).

[27] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu: A Memory-efficient Parallel String Matching Architecture for High-speed Intrusion Detection, IEEE Journal on Selected Areas in Communications, vol. 24, pp. 1793-1804 (2006).