MDH: A High Speed Multi-phase Dynamic Hash String Matching Algorithm for Large-Scale Pattern Set

Zongwei Zhou[1,2], Yibo Xue[2,3], Junda Liu[1,2], Wei Zhang[1,2], and Jun Li[2,3]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, China
[2] Research Institute of Information Technology, Tsinghua University, Beijing, China
[3] Tsinghua National Laboratory for Information Science and Technology, Beijing, China
zhou-zw02@mails.tsinghua.edu.cn

Abstract. String matching algorithm is one of the key technologies in numerous network security applications and systems. Nowadays, the increasing network bandwidth and pattern set size both calls for high speed string matching algorithm for large-scale pattern set. This paper proposes a novel algorithm called Multi-phase Dynamic Hash (MDH), which cut down the memory requirement by *multi-phase hash* and explore valuable pattern set information to speed up searching procedure by *dynamic-cut heuristics*. The experimental results demonstrate that MDH can improve matching performance by 100% to 300% comparing with other popular algorithms, whereas the memory requirement stays in a comparatively low level.

Keywords: Network Security, String Matching Algorithm, Multi-Phases Hash, Dynamic-Cut Heuristics.

1 Introduction

Along with the rapid development of modern network technology, demands for anti-attack and security protection are now facing a drastic increase in almost all network applications and systems. String matching is one of the key technologies of them. For example, widely deployed network intrusion detection and prevention systems (NIDS/IPS) often use signature-based method to detect possible malicious attacks, so string matching algorithm is their basic operation. It has been demonstrated that string matching takes about 31% of the total processing time in Snort[1][5], the most famous open source NIDS system[8]. The other remarkable instance is content inspection network security systems. More and more such applications, including, but not limited to, anti-virus, anti-spam, instant message filtering, and information leakage prevention require payload inspection as a critical functionality. And, string matching is also the most widely used technology in payload scanning.

However, string matching technology now encounters new challenges from two important facts, both of which indicate that more efficient and practical high speed string matching algorithms for large-scale pattern set are urgently needed.

The first challenge is that large-scale pattern sets are becoming increasingly pervasive. In this paper, we define pattern set that has more than 10, 000 patterns as large-scale pattern set, in contrast to small or middle size pattern sets in typical

S. Qing, H. Imai, and G. Wang (Eds.): ICICS 2007, LNCS 4861, pp. 201–215, 2007.

network security systems. As more types of virus, worm, trojan and malware spread on the Internet, pattern set size in anti-virus applications keeps increasing. For example, the famous open source anti-virus software—Clam AntiVirus[2] now has more than 100,000 patterns, and daily update is still quickly enlarging it. From February 14th to March 18th, 2007, the pattern set size increase by about 10, 000. However, most existing string matching algorithms are designed and tested under small and moderate pattern set. They cannot be efficiently used in large-scale scenario.

Secondly, network edge bandwidth is increasing from 100Mbps to 1Gbps or even more. Such development demands for high throughput of current inline network security applications. In newly emerging UTM (Unified Threat Management) systems, turning on real-time security functionalities like intrusion prevention, anti-virus, and content filtering will greatly reduce the system overall throughput, because such functionalities all need extensive string matching operation. However, string matching algorithms now are still far from efficient enough to meet the needs driven by bandwidth upgrade.

This paper proposes a novel high-speed string matching algorithm, Multi-Phase Dynamic Hash (MDH), for large-scale pattern sets. We introduce *multi-phase hash* to cut down the memory requirement and to deal with high hash collision rate under large-scale pattern set. And we also propose a novel idea, *dynamic-cut heuristics*, which can explore the independence and discriminability of the patterns to speed up the string matching procedure. Experimental results of both random pattern sets and some real-life pattern sets show that MDH increases the matching throughput by about 100% to 300%, compared with some other popular string matching algorithms, whereas, maintain its memory requirement at a low level.

The rest of this paper is structured as follows: Section 2 overviews pervious work on string matching algorithms. Section 3 describes in detail our MDH algorithm. The experimental results are given out in Section 4 to demonstrate high matching performance and low memory requirement of our algorithm. Conclusions and future work are in the last section.

2 Related Work

There are basically two categories of string matching algorithms—*forward algorithm* and *backward algorithm*. They both use a window in the text, which is of the same length as the pattern (the shortest pattern if there are multiple patterns). The window will slide from leftmost of the text to the rightmost. *Forward algorithm* examines the characters in the text window from left to right, while *backward algorithm* starts at the rightmost position of the window and read the characters backward.

Among the *forward algorithms*, Aho-Corasick algorithm[6] is the most famous one. This algorithm preprocesses multiple patterns into a deterministic finite state automaton. AC examines the text one character at a time, so its searching time complexity is $O(n)$ when n is the total length of the text. This means that AC algorithm is theoretically regardless of pattern numbers. However, in practical usage,

automaton size increases quickly when the pattern set size goes up, which would require too much memory. This limits the scalability of AC to large-scale string matching.

It has been demonstrated that *backward algorithm* have higher average search speed than *forward algorithm* in practical usage, because it can skip unnecessary character comparisons in the text by certain heuristics[3]. Boyer-Moore algorithm [7] is the most well-known *backward algorithm* used in single pattern matching. There are two important heuristics in BM algorithm, *bad character* and *good suffix*, which is shown in Fig.1. BM calculates both of the shift values according to these two heuristics and then shifts the window according to the bigger one.

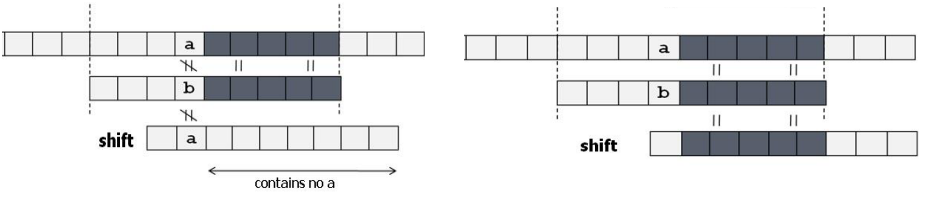

Fig. 1. Bad character (left) and good suffix (right) heuristic, *y* denotes the text and *x* is the pattern. *u* is the match suffix of the text window.

Wu-Manber algorithm[4] extended BM to concurrently search multiple strings. Instead of using *bad character* heuristic to compute the shift value, WM uses a character block including 2 or 3 characters. WM stores the shift values of these blocks in SHIFT table and builds HASH table to link the blocks and the related patterns. The SHIFT table and the HASH table are both hash tables which enable efficient search. Moreover, in order to further speed up the algorithm, WM also builds another hash table, the PREFIX table, with the two-byte prefixes of the patterns. This algorithm has excellent average time performance in practical usage. But, its performance is limited by minimum pattern length *m* since the maximum shift value in SHIFT table equals to *m*-1.

However, when pattern set is comparatively large, the average shift value in WM algorithm will decrease and thus the searching performance will be compromised. B. Xu and J. Li proposed the Recursive Shift Indexing (RSI)[10] algorithm for this problem. RSI engages a heuristic with a combination of the two neighboring suffix character blocks in the window. It also uses bitmaps and recursive tables to enhance matching efficiency. These ideas are enlightening for large-scale string matching algorithms.

J. Kytojoki, L. Salmela, and J. Tarhioin also presented a q-Grams based Boyer-Moore-Horspool algorithm[11]. This algorithm cuts a pattern into several q-length blocks and builds q-Grams tables to calculate the shift value of the text window. This algorithm shows excellent performance on moderate size of pattern set. However, when coming into large-scale scope, it is not good enough both in searching time and memory requirement.

C. Allauzen and M. Raffinot introduced Set Backward Oracle Matching Algorithm (SBOM)[12]. Its basic idea is to construct a more lightweight data structure called

factor oracle, which is built only on all reverse suffixes of minimum pattern length *m* window in every pattern. It consumes reasonable memory when pattern set is comparatively large.

There are also some other popular *Backward algorithms* which combine the BM heuristic idea and AC automaton idea. C. Coit, S. Staniford, and J. McAlerney proposed AC_BM algorithm[8]. This algorithm constructs a prefix tree of all patterns in preprocessing stage, and then takes both BM bad character and good suffix heuristics in shift value computation. A similar algorithm called Setwise Boyer Moore Horspool (SBMH)[9] is proposed by M. Fisk and G. Varghese. It utilizes a trie structure according to suffixes of all patterns and compute shift value only using the bad character heuristic. However, these two algorithms are also limited by the memory consumption when the pattern set is large.

3 MDH Algorithm

We have reviewed some popular multiple string matching algorithms. They are the best algorithms under different circumstances. But, for large-scale pattern sets, all of them suffer drastic matching performance decline. Some of them, such as AC, AC_BM and SBMH, also face memory explosion. Moreover, as we have considered, there are few algorithms now solve the large-scale pattern set problem well. In this context, MDH is designed to both improve the matching performance and maintain moderate memory consumption. Based on WM algorithm, our new algorithm has two main improvements:

First, when pattern sets become larger, WM algorithm has to increase the size of the SHIFT table and the HASH table to improve matching performance. This would consume lots of memory. MDH introduces *multi-phase hash* to cut down the high memory requirement.

Second, WM algorithm considers only the first *m* characters of the patterns. It is simple and efficient, but overlooks helpful information in other characters. Therefore, MDH introduces *dynamic-cut heuristics* to select the optimum *m* consecutive characters for preprocessing. This mechanism will bring in higher matching performance.

3.1 Key Ideas of MDH

In the following description, we let *B* to be the block size used in WM and MDH, *m* to minimum pattern length, $\sum$ to be the alphebet set of both pattern and text, $|\sum|$ to be the alphebet set size, *k* to be the total pattern number, *l* to be the average length of all the patterns.

3.1.1 Multi-phase Hash

In WM algorithm, a certain SHIFT table entry stores the minimum shift value of all the character blocks hashed to it. As the pattern number increases, high hash collision

will reduce the average shift value $E(shift)$ in SHIFT table and thus compromise the matching performance.

Therefore, a better algorithm for large-scale pattern set always increases character block size B to deal with the high hash collision rate. But larger B will result in bigger SHIFT and HASH table, and thereby greatly increases the memory requirement. Considering the limited cache in modern computers, high memory consumption will decline the cache targeting rate and increase average memory access time. It will in turn decrease the matching performance. On the other hand, it is also difficult to load such large data structures into SRAM when the algorithm is implemented on current high speed appliance such as network processor, multi-thread processing chips and FPGA. This will limit its scalability to hardware implementations.

Under such observations, we propose a novel technique called *multi-phase hash*. In WM algorithm, general hash function is used to build SHIFT table and HASH table, the character blocks and the hash table entries are one-to-one correspondent. But in MDH, we use two compressed hash table, the SHIFT table and the PMT table, to replace them. They are of the similar functionality, but consume less memory. MDH first choose a compressed hash function h_1, to reduce SHIFT table from $|\sum|^B$ entries to $|\sum|^{a/8}$ ($a < 8B$), which means that h_1 only uses a bits of the B-length character block. However, compressing the SHIFT table entries together will also reduce the average shift value, similar with increasing pattern set size. Some entries with non-zero shift value would be hashed into zero shift value entry. This will bring in more character comparison time in matching procedure. So we then introduce another compressed hash table, PMT table, to separate the non-zero shift value entries away from zero shift value entry. When a certain character block with non-zero shift value is hashed into a zero shift value entry, MDH uses another hash function h_2 to rehash it and store their shift value as *skip* value in the PMT table. PMT table is of the size $|\sum|^{b/8}$ ($b < a < 8B$). Moreover, PMT table also linked by some possible matching patterns, similar with HASH table in WM. The number of these pattern linked to a certain PMT table entry is recorded as its *num* value.

3.1.2 Dynamic-Cut Heuristics

Following the common practice of some previous work[3], the average character comparison times $E(comparison)$ is important for the matching performance of WM algorithm. Large-scale pattern set can increase $E(comparison)$ and compromise the matching performance. We handle it by introduce *dynamic-cut heuristics*. Mathematical analysis of $E(comparison)$ decides the detail mechanisms used in dynamic-cut heuristics.

Let ***ZR*** to be the ratio of the number of *zero entries* (entries with zero shift value) SHIFT entry to the total number of SHIFT table entries. Let T_0 to be the number of *non-zero entries* (entries linked with possible matching patterns) in PMT table, therefore k / T_0 is the *average number of possible matching patterns* (**APM**) in PMT table.

In the searching stage, MDH first checks the shift value in SHIFT table. If it is zero, the algorithm then checks the *skip* value in PMT table. Only if the *skip* value is also zero should the algorithm verify the possible matching patterns. So the probability of comparison times equals to x ($\Pr(comparison = x)$) is calculated as follows:

$$\begin{cases} \Pr(comparison = 1) = 1 - ZR \\ \Pr(comparison = 2) = ZR * (1 - T_0 / |\sum|^{b/8}) \\ \Pr(comparison > 2) = ZR * T_0 / |\sum|^{b/8} \end{cases} \tag{1}$$

Thus, under average condition, $E(comparison)$ could be estimated as follows:

$$\begin{aligned} E(comparison) = 1 * \Pr(comparison = 1) + 2 * \Pr(comparison = 2) \\ + (2 + l * APM) * \Pr(comparison > 2) \end{aligned} \tag{2}$$

From (1) and (2), we get:

$$E(comparison) = 1 + ZR + l * k * ZR / |\sum|^{b/8} \tag{3}$$

Moreover, the above analysis is only under the normal condition of network security application, when the pattern matches in the text are comparatively sparse. However, new denial-of-service attacks, such as sending text of extremely high matches and jamming the pattern matching modules, have emerge to compromise the network security application with BM-family string matching algorithms. Thus it is very necessary to consider the condition of heavy-load case or even worst-case, when there are lots of matches in the text. Under such circumstance, $E(comparison)$ will be calculated as follows:

$$E_w(comparison) \approx 2 + l * APM \tag{4}$$

Therefore, after setting the SHIFT table size and PMT table size in *multi-phase hash*, there still remains two probabilities for improving the searching performance. First, from equation (3), smaller ***ZR*** results in smaller $E(comparison)$ under normal condition and thereby brings in higher average searching performance. Secondly, as in equation (4), smaller ***APM*** results in smaller $E_w(comparison)$ and thus ensures high searching performance for worse-case condition.

According to the above analysis, MDH uses *dynamic-cut heuristics* to *cut* every pattern into the optimum consecutive *m* characters and to reduce the ***ZR*** and ***APM*** in SHIFT table and PMT table. Theoretically, MDH could compute all the ***ZR*** and ***APM*** values under all the cutting conditions and then choose the optimum one. Apparently, such heuristic mechanism demands for high time and memory consumption in preprocessing when the pattern number k and average pattern length l are large. Note that in most network security application and systems with large-scale string matching, such as anti-virus and content inspection, pattern sets are changing very fast. It is improper to choose such complex preprocessing mechanism.

Thus we implement the heuristics in a comparatively simple way, which is described detail in the following section.

3.2 Algorithmic Details of MDH

3.2.1 Preprocessing Stage

In the following description, we let the block length B=4, SHIFT table size a=20, PMT table size b=17. The pattern set is {opionrate, torrential, extension, cooperation}. So the minimum pattern length m=9. << denotes for the bit operator of left shift. Hash function h_1 and h_2 are as follows:

$$h_1(\text{block})=(*(\text{block}))\&0x000FFFFF \tag{5}$$

$$\begin{aligned} h_2(\text{block})=&((*(\text{block})<<12)+(*(\text{block}+1)<<8)\\ &+(*(\text{block}+2)<<4)+*(\text{block}+3))\&0x0001FFFF \end{aligned} \tag{6}$$

There are three steps in preprocessing stage:

Step1: Initialize SHIFT table and PMT table, set all *shift* value and *skip* value to be *m-B*+1, all *num* value to be zero. Each pattern has its *offset* value, that is, the offset of optimum *m* window in the pattern. All *offset* value is initiated to zero.
Step2: Process the patterns one by one, set their optimum *m* window position according to the *dynamic-cut heuristics* and note down the *offset* value. Meantime, all the suffix character blocks of these windows are added into the SHIFT table and the PMT table. Related *shift* value and *num* value are set.
Step3: Process the patterns one by one again, add the other blocks (except the suffix block) in all the optimum *m* windows into the SHIFT table and the PMT table. Related *shift* value and *skip* value are set.

3.2.1.1 Step2—Optimum m Window Position Setting. In this step, the algorithm processes the patterns one pattern by another and calculates their optimum m window position.

"opionrate" is the one of the shortest patterns in the pattern set. So its optimum *m* window is "opionrate" itself. Its suffix block "rate" is added into SHIFT table and PMT table. The algorithm sets *Shift* value in the $h_1(\text{rate})$ SHIFT entry to 0, set *num* value in the $h_2(\text{rate})$ PMT entry to 1, and link the pattern after $h_2(\text{rate})$ PMT entry.

For pattern "torrential", it has two possible *m* window positions—"torrentia" and "orrential". The algorithm check the $h_1(\text{ntia})$ SHIFT entry, the shift value is 4. Then we check the $h_1(\text{tial})$ SHIFT entry, this shift value is still 4. So optimum *m* window is not found, the algorithm will manually set "torrentia" as the optimum *m* window and set related *shift* and *num* value. The procedure of adding the pattern "extension" is similar with that of adding "opionrate" because they are both the shortest patterns. Then here comes the last pattern "cooperation". The procedure of adding this pattern reveals the effect of *dynamic-cut heuristics*. There are three possible *m* window positions—"cooperati", "ooperatio" and "operation". The algorithm first checks the $h_1(\text{rati})$ SHIFT entry and found its shift value is 4, then checks the $h_1(\text{atio})$ SHIFT entry and gets the same result. So, the algorithm moves the window again and checks the $h_1(\text{tion})$ SHIFT entry. Since $h_1(\text{tion})=h_1(\text{sion})$, its shift value will be zero. Note

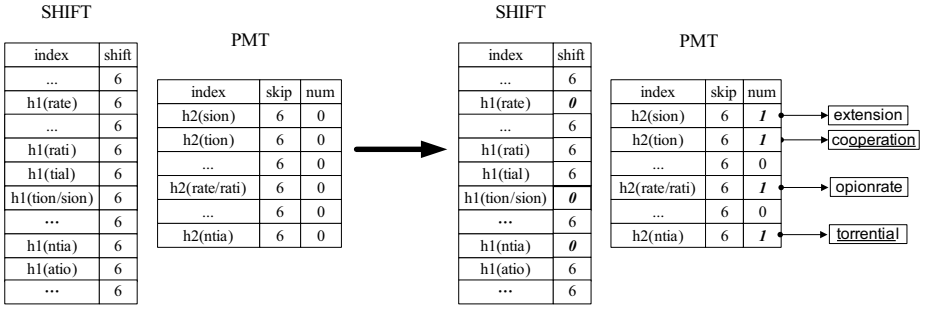

Fig. 2. SHIFT table and PMT table before and after setting optimum minimum *m* window position for pattern set {opionrate, torrential, extension, cooperation}

that the h_2(tion) PMT entry has a zero *num* value. According to our heuristics, "operation" will be the optimum *m* window of pattern "cooperation" and the related *offset* value is 2.

Figure 2 illustrates that, without *dynamic-cut heuristics*, the *shift* value of h_1(rati) SHIFT entry will be zero, there would be four SHIFT entries with zero *shift* value and therefore the **ZR** becomes bigger. And also "cooperation" and "opionrate" will both be linked to h_2(rate) PMT entry and **APM** becomes larger, since h_2(rate)=h_2(rati). Thus, it is demonstrated that *Dynamic-cut heuristics* helps to make both **ZR** and **APM** smaller, which will contribute to bring in higher searching performance. Comparison experiments between MDH without *dynamic-cut heuristics* and MDH full implementation will appear in Section 4 to further prove its effect.

3.2.1.2 Step3—Adding Characters Blocks in the optimum m windows. In this step, we take processing pattern "opionrate" for example. The algorithm put a *B*-length block window (*B* window) at the leftmost position of the pattern and slide. Let j to be the offset of *B* window, the shift value of the character block in *B* window can be calculated by *m-B-j*. First compute the hash value of "opio" by hash function

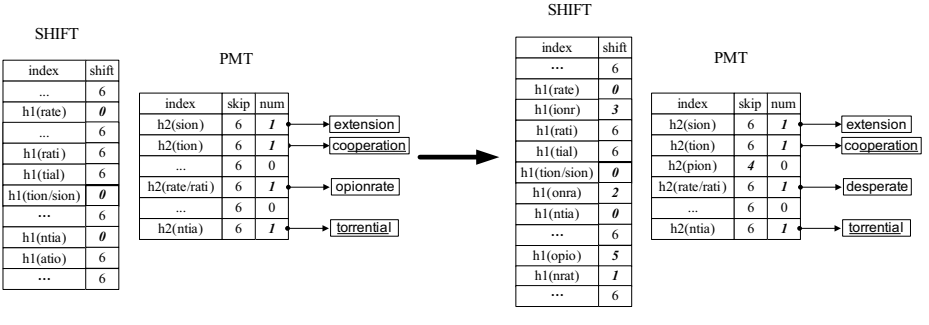

Fig. 3. SHIFT table and PMT table before and after filling shift value and skip value of all *B*-length character blocks in pattern "opionrate"

h_1. The *shift* value in h_1(opio) SHIFT entry is 6, and the shift value of "opio" is 5. So the algorithm will note down the smaller value 5 as the new *shift* value of this entry. Then we will compute the hash value of "pion", which is the same as h_1(tion/sion). The shift value of h_1(tion/sion) entry is zero. Under this condition, the algorithm will compute h_2(pion) and index to the related PMT entry. The *skip* value of h_2(pion) PMT entry is 6 and the shift value of "pion" is 4. So the algorithm will note down the smaller value 4 as the new *skip* value of this entry. Following this way, the algorithm then processes character block "ionr", "onra", "nrat".

Figure 3 shows the SHIFT table and the PMT table before and after the whole procedure above. Apparently, without *multi-phase hash* idea, character block "pion" will be hashed into h_1(tion/sion) SHIFT entry with zero *shift* value. It will cause unnecessary pattern verification of "cooperation" with a suffix block "tion". However, the algorithm will get its real shift value by checking the *skip* value of h_2(pion) PMT entry and unnecessary character comparison can be avoided.

3.2.2 Scanning Stage

The scanning procedure is comparatively simple and explicit. *B*-length text window slides from leftmost position of the text to right. Each time we examine *B* characters in the text window, calculates its hash value according to hash function h_1, check the relevant SHIFT table entry. If the shift value in this entry is not zero, move the text rightwards by the shift value and restart this procedure. Otherwise, hash this text block again using hash function h_2, use the new hash value to index to the corresponding PMT table entry. Verify every possible matching pattern linked in this entry using naïve comparison method. After that, move the text rightwards by the skip value of this entry and restart the whole procedure.

4 Experimental Results

This section gives out a serial of experiments to demonstrate the performance of MDH algorithm. The test platform is a personal computer with one dual-core Intel Centrino Duo™ 1.83GHz processor and 1.5GB DDR2 667MHz memory. The CPU has 32KB L1 instruction cache and 32KB L1 data cache. The shared L2 cache is 2048KB.

The text and patterns are both randomly generated on alphabet set $|\sum| = 256$. And we then insert all the patterns into random position of the text for three times to guarantee a number of matches between random text and patterns. In the first experiment of searching time comparison, we also use a recent antivirus pattern set from Clam AntiVirus to demonstrate the practical performance of MDH algorithm. The text size in the following tests is 32MB. The pattern length of our large-scale pattern sets extends from 4 to 100 and 80% of patterns are of the length between 8 and 16, which is comparatively close to content inspection based network security application such as instant message filtering and content inspection, recommend by CNCERT/CC [13].

4.1 Searching Time and Memory Requirement Comparison

To better evaluate the performance of MDH, we choose five typical multiple string matching algorithms which are widely deployed in recent practical applications. The source codes of AC, AC_BM, WM algorithms are adopted from Snort. Unnecessary codes about case sensitive related operations are eliminated to take off extra time and memory consuming. In WM algorithm, we set the block length B=2. The source codes of SBMH and SBOM are from [14].

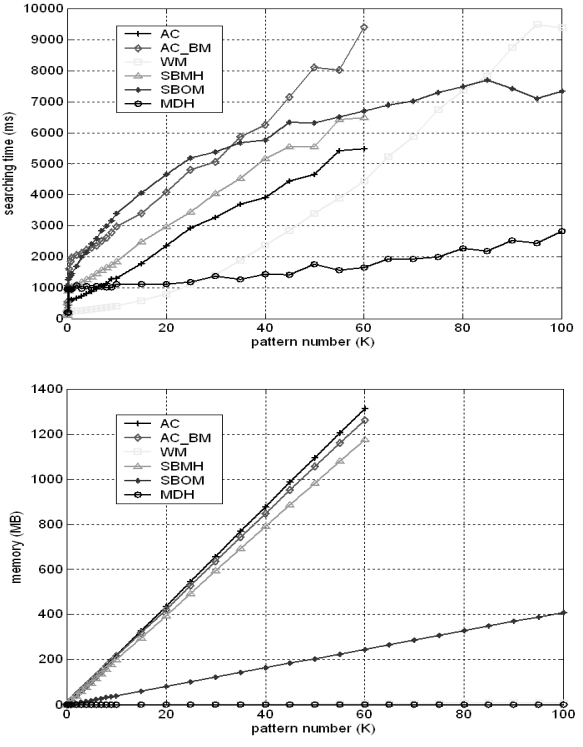

Fig. 4. The upper graph is the searching time comparison between MDH and some typical algorithms. Under the pattern sets larger than 30k, MDH is much better than any other algorithms in this experiment. And the scalability of MDH to even larger patter sets more than 100k is promising since its performance decline is not so rapid as other algorithms when pattern set size increases from 10k to 100k. The lower graph is the memory comparison. Table-based algorithm like MDH and WM algorithm consume much less memory than other algorithms in the experiment.

Figure 4 illustrates that the performance of all the five typical algorithms suffer drastic declines when pattern set size exceeds 30k. Their matching throughput is fewer than 96Mbps with 50k patterns. Algorithms like AC, AC_BM and SBMH can not support pattern sets larger than 60k under our test condition because of their high memory consumption. When there are 100k patterns, the matching throughput of

MDH algorithm is still more than 100Mbps. It exceeds SBOM by 169% and WM by 231%. In addition, MDH algorithm possesses high stability as pattern set size increases and also excellent scalability to small and moderate pattern set size. The stable performance also indicates that it has better scalability to supper-large-scale pattern sets. In our further test, the matching throughput of MDH is abut 48.8 Mbps when pattern set size is 200k, still better than that of WM and SBOM under 100k pattern set.

MDH algorithm is also superior in memory requirement. When pattern set size increases up to 50k, memory requirement of all algorithms except WM and MDH are more than 200MB. Table-based algorithms like WM and our solution only consume less than 20 MB memory even in 100k pattern sets.

4.2 Experiments on Real-Life Pattern Set

To demonstrate the practical performance of MDH algorithm, we choose the real-life pattern set used in Clam AntiVirus in this experiment. The total number of the current virus data base has 102, 540 patterns. We removed all the patterns that is either represented by regular expressions or of the length shorter than 4. After that, the pattern set size is 77, 607. We also form three different subset of the size 20k, 40k and 60k. The minimum pattern length of all these four pattern sets is 4. SBOM and WM are chosen to be compared with MDH, because these two algorithms also have reasonable searching time performance and memory consumption in Section 4.1.

Table 1. In this table, Mem represents the total memory consumption and Thr denotes the matching throughput, that is, size of the text that have been processed in a second Under large-scale pattern sets. Under Clam AntiVirus pattern set, MDH possesses both higher searching performance and lower memory consumption when comparing with WM and SBOM algorithm.

Algorithm	20k		40k		60k		77k	
	Thr (Mbps)	Mem (MB)	Thr (Mbps)	Mem (MB)	Thr (Mbps)	Mem (MB)	Thr (Mbps)	Mem (MB)
MDH	250.56	3.82	203.28	5.2	174.24	8.08	150.16	10.41
WM	329.52	3.33	126	5.2	66.88	8.53	43.36	11.27
SBOM	69.68	81.87	56.16	162.5	43.76	244.7	36.48	316.84

From Table 1, we can see, from 20k to 77k patterns, the searching throughput of MDH algorithm does not suffer drastic decline as WM and SBOM algorithm. This stable performance indicates that MDH has better scalability to even supper-large-scale pattern sets in real-life applications. When there are 77k patterns, the matching throughput of MDH algorithm is more than 150Mbps, which exceeds SBOM by 311% and WM by 246%. Meanwhile, MDH only consumes about 3 to 11 MB memory to process these pattern sets, no more than WM algorithm and much fewer than SBOM algorithm. It is fair to assert that MDH algorithm possesses excellent time and space performance under the large-scale pattern sets from real-life security applications.

4.3 Experiments on Multi-phase Hash

Table 2 is the result of comparison test between WM algorithm (*B*=2), WM algorithm (*B*=3) and MDH with *multi-phase hash*. In this table, **MEM** stands for the total memory used in WM or MDH algorithm. When pattern number is more than 10k, **ZR** becomes very high in WM algorithm (*B*=2). According to equation (3) in Section 3.1.2, higher **ZR** would bring in bigger $E(comparison)$ and greatly compromise the searching performance. If *B*=3, **ZR** becomes comparatively low to ensure good searching performance. However, under this condition, SHIFT table and HASH table will become bigger since these tables are both of the size $|\sum|^B$. So **MEM** in WM (*B*=3) increase to more than 80MB. With *multi-phase hash,* MDH is able to maintain moderate **ZR**. Its **MEM** is nearly in the same level with WM (*B*=2) and only about 2%~7% of WM (*B*=3).

Table 2. This table is a comparison of **ZR** and **MEM** between WM algorithm (*B*=2), WM algorithm (*B*=3) and MDH algorithm with *multi-phase hash*. **ZR** is high in WM algorithm (*B*=2) under large-scale pattern set. If *B*=3, WM algorithm possesses low **ZR**, but another problem is that it consumes too much **MEM**. MDH is both good in maintaining low **ZR** and resonable **MEM**.

Pattern number	WM(*B*=2)		WM(*B*=3)		MDH	
	ZR (%)	MEM (MB)	ZR (%)	MEM (MB)	ZR (%)	MEM (MB)
10k	14.2	0.95	0.059	80.64	0.85	2.42
25k	31.7	1.91	0.149	81.59	1.91	2.98
50k	53.3	3.5	0.297	83.19	3.46	3.93
75k	68.0	5.09	0.446	84.78	4.32	4.87
100k	78.3	6.69	0.594	86.38	6.25	5.81

4.4 Experiments on Dynamic-Cut Heuristics

From Table 3, we can see that **ZR** has a drastic decline when dynamic-cut heuristics are applied. In 10k pattern set, dynamic-cut heuristics reduce the zero entry number by about 10%, and in 100k patter set, this number increases up to nearly 30%. The heuristics' influence on **ZR** becomes more significant when pattern set size is larger. It also has been demonstrated that **APM** value becomes comparatively smaller owing to dynamic-cut heuristics.

As for time performance, dynamic-cut heuristics save about 7.6% to 14% searching time when pattern number ranges from 10k to 100k. Noticeably, the bigger the pattern set is, the more significant the time-saving effect will be. It strongly testifies the excellent scalability of the dynamic-cut heuristics to even larger pattern set. However, the overhead in processing time is still reasonable since most of the network security applications do not have high frequency of pattern set changing and more attentions are focused on improving the searching time.

Table 3. ZR is the zero SHIFT entry radio, the same as in Table 2. APM indicates the average number of possible matching patterns in PMT table. MP denotes of the MDH implementation without dynamic-cut heuristics. We can see that dynamic-cut herurisitcs have greatly reduce the Znum and AN in PMT, which contributes to the searching time decrease.

Pattern Number	ZR		APM		Preprocessing Time(ms)		Searching Time (ms)	
	MP	MDH	MP	MDH	MP	MDH	MP	MDH
10k	9940	8878	1.04	1.03	18.8	20.1	1112	1028
30k	29458	23391	1.12	1.08	26.4	37.4	1459	1312
50k	48461	36262	1.2	1.12	36	62.8	1668	1512
70k	67005	48198	1.29	1.16	49.9	84.4	2118	1877
100k	93842	65494	1.43	1.22	68.3	105.9	3117	2680

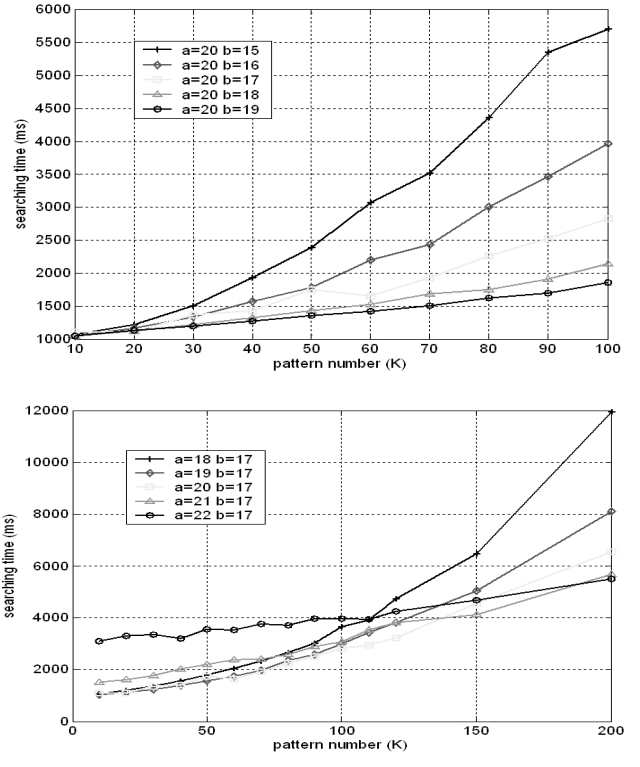

Fig. 5. In the upper graph, SHIFT table size is set to 2^{20} (a=20) and the PMT table size is ranging from 2^{15} (b=15) to 2^{19} (b=19). MDH has less run time (or better performance) when using larger PMT table size. The experiment related with the lower graph is done under same PMT table size as 2^{17} (b=17). SHIFT table size is ranging from 2^{18} (a=18) to 2^{22} (a=22). The optimum SHIFT table size is different under different pattern sets.

4.5 SHIFT and PMT Table Size Selection

The selection of SHIFT and PMT table size is the critical part of MDH implementation. In the upper graph of Fig 5, we can conclude that bigger PMT table is more helpful in improving searching performance. It matches our previous analysis. When PMT table is larger, we are able to partition all character blocks with zero SHIFT value into more entries. So **APM** value could be smaller. This would highly reduce unnecessary verification time and benefit for final performance. Thus, within the memory limitation, it is better to choose as larger PMT table as possible. In MDH algorithm, we choose a moderate and acceptable PMT table size as 2^{17} (b=17).

In the lower graph of Fig 5, we test the selection of SHIFT table size under the same PMT table size of 2^{17} (b=17). The optimum SHIFT table size is related to the pattern set size. From 10k to about 110k patters, MDH with SHIFT table size of 2^{19} and 2^{20} are of higher searching speed than other ones. And for pattern set between 110k and 190k, a=21 becomes the best choice. When pattern number increases to 200k or even more, a=22 will perform better than others. Moreover, we can also conclude that the run times curve of larger SHIFT table size always possess smaller average slope. The reason is that in large SHIFT table, **ZR** is comparatively small. The pattern set increment can not significantly raise this ratio and compromise the matching performance.

Thus, we may conclude that the selection of SHIFT table size depends on the pattern set size. The algorithm should choose larger SHIFT table size to meet the needs of larger patter set. In this paper, we focus on pattern sets ranging from 10k to 100k and thus set the SHIFT table size to be 2^{20} (a=20).

5 Conclusion and Future Works

This paper proposes a novel string matching algorithm named Multi-Phases Dynamic Hash algorithm (MDH) for large-scale pattern set. Owing to *multi-phase hash* and *Dynamic-cut heuristics*, MDH can improve matching performance under large-scale pattern set by about 100% to 300% compared with other typical algorithms, whereas the memory requirement remains at a comparatively low level. Low memory requirement will help to raise the cache targeting rate in practical usage and thereby improve the matching performance. It would also contribute to support accelerating hardware architectures based on MDH, like FPGA and new multi-core chips.

However, several works will be considered in the future. We are in the progress of finding the relationships between character block B, SHIFT table size a, PMT table size b and pattern sets size k through more experimental and mathematic analysis. We can also study more complex and efficient alternatives for *dynamic-cut heuristics*. In addition, architecture design of network content filtering systems based on MDH and multi-thread models will also be within our scope.

Acknowledgement. The authors thank CNCERT/CC for their support of this work. CNCERT/CC is the abbreviation of National Computer Network Emergency Response Technical Team/Coordination Center of China. The authors would also like

to thank Mr. Kai Li, Mrs. Xue Li, Mr. Bo Xu, Mr. Xin Zhou and Mr. Yaxuan Qi for enlightened suggestions and helps. Last but not least, the authors would like to thank numerous volunteers who contributed to the open source projects like Snort and Clam AntiVirus.

References

1. Roesch, M.: Snort: lightweight intrusion detection for networks. In: Proc. of the 1999 USENIX LISA Systems Administration Conference (1999)
2. Clam AntiVirus™ http://www.clamav.net/
3. Navarro, G., Raffinot, M.: Flexible pattern matching in strings. Cambridge University Press, Cambridge (2002)
4. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching, Technical Report TR-94-17, Department of Computer Science, University of Arizona (1994)
5. Snort, http://www.snort.org/
6. Aho, A., Corasick, M.: Fast pattern matching: an aid to bibliographic search. Journal on Communication ACM 18(6), 333–340 (1975)
7. Boyer, R., Moore, J.: A fast string searching algorithm. Journal on Communication. ACM 20(10), 762–772 (1977)
8. Coit, C., Staniford, S., McAlerney, J.: Towards faster string matching for intrusion detection or exceeding the speed of snort, DARPA Information Survivability Conference and Exposition, pp. 367–373 (2001)
9. Fisk, M., Varghese, G.: An analysis of fast string matching applied to content-based forwarding and intrusion detection. Technical Report CS2001-0607 (updated version), University of California-San Diego (2002)
10. Xu, B., Zhou, X., Li, J.: Recursive shift indexing: a fast multi-pattern string matching Algorithm. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, Springer, Heidelberg (2006)
11. Kytojoki, J., Salmela, L., Tarhio, J.: Tuning string matching for huge pattern sets? In: Baeza-Yates, R.A., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 211–224. Springer, Heidelberg (2003)
12. Allauzen, C., Raffinot, M.: Factor oracle of a set of words, Technical report 99-11, Institut Gaspard-Monge, Universite de Marne-la-Vallee (1999)
13. National Computer Network Emergency Response Technical Team/Coordination Center of China, http://www.cert.org.cn/
14. Network Security Lab: Research Institute of Information Technology, Tsinghua University, Beijing, http://security.riit.tsinghua.edu.cn/share/pattern.html