Recursive Shift Indexing: A Fast Multi-Pattern String Matching Algorithm

Bo Xu, Xin Zhou, and Jun Li

Department of Automation, Tsinghua University, Beijing, China

Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China

Tsinghua National Lab for Information Science and Technology, Beijing, China

xb00@mails.tsinghua.edu.cn

Abstract. String matching algorithms are essential for network devices that filter packets and flows based on their payload. Applications like intrusion detection/prevention, web filtering, anti-virus, and anti-spam all raise the demand for efficient algorithms dealing with string matching. This paper presents a novel multi-pattern string matching algorithm which reduces character comparisons based on recursive shift indexing. Theoretical analysis and experimental results show that the new algorithm is highly efficient: Its search time is cut down significantly compared with other popular existing algorithms, whereas its memory occupation stays at a low level. It is also demonstrated that the proposed algorithm has a simpler structure for easy implementation.[1]

Keywords: string matching, shift indexing, intrusion detection, intrusion prevention

1 Introduction

There is increasing types and numbers of malicious attacks that attempts to compromise the integrity, confidentiality, availability, and other security aspects of computers and networks. To complement other products, such as firewalls and VPNs, network intrusion detection system (NIDS) and network intrusion prevention system (NIPS) products have been developed and deployed to defend computer and networking systems [1]. These systems inspect the ordered characters in the payload of packets or flows against attacks, in addition to protocol analysis, statistical analysis, and other mechanisms. When it is used in sniffing or taping mode, it is normally called

[1] This work was supported by the Intel IXA University Program and research grant from Juniper Networks.

NIDS that just reports the intrusion, and the actions will be taken by administrator or firewall. When it is used in inline mode, it is usually referred as NIPS that reports and responds to abnormal behavior on the spot. In this paper, the algorithms to be discussed can be used in both NIDS and NIPS, thus the term *network intrusion management system* (NIMS) is used to include both of them as well as other systems with similar functionalities.

The string matching in NIMS is based on a set of patterns, also referred as intrusion signatures, each indicating an intrusion threat. NIMS inspects the network traffic to find out whether such patterns occur in the payload. In Snort [2], a well known open source NIMS, it has been reported that as much as 31% of total processing time is due to string matching [3]. It is also reported that in the case of Web-intensive traffic, this computational cost can be as much as 80% of the total processing time [4]. Meanwhile, the ever increasing bandwidth requirement and the growing number of potential threats also call for high performance NIMS to keep up with the demand. These motivations advocate the research on more efficient string matching algorithms.

In this paper, a novel multi-pattern string matching algorithm is proposed to achieve high performance.

2 Previous Work

The best known algorithms for string matching are BM [5] proposed by R. Boyer and J. Moore in 1977, and AC [6] proposed by A. Aho and M. Corasick in 1975. Being a single pattern string matching algorithm, the basic idea of BM is to make use of *bad character* and *good suffix* to generate leaps so that character comparisons can be skipped at some positions. When dealing with multiple patterns at high speed, BM will not be efficient enough since it has to perform iterative searching for each pattern with a time complexity of $O(mn)$, where m is the number of patterns and n is the length of the payload text. AC is a multi-pattern string matching algorithm that adopts a finite state automaton to organize multiple pattern strings in order to get searching of multiple patterns done in one pass. Its time complexity is $O(n)$, regardless of the number of patterns. However, AC suffers an intrinsic deficiency for high speed: it does not excavate any heuristics to avoid unnecessary character comparisons.

S. Wu and U. Manber introduced the WM algorithm [7] in 1994, which mainly engages the bad character heuristic of BM for multi-pattern string matching. WM uses two or three suffix characters as a block to generate leaps instead of using a single character. Once the suffix character block hit a match, the algorithm eliminates ineligible patterns according to the prefixes, through a hash table. Finally, the naïve comparison is applied to verify whether the remaining patterns exist in the text.

M. Fisk and G. Varghese proposed the Setwise Boyer-Moore-Horspool (Setwise BMH) algorithm[3] in 2002, which constructs a trie that stores all patterns based on the suffixes, and creates a leap table based on the bad character heuristic. The compact trie will get rid of many unnecessary comparisons and hence improve the matching efficiency. When mismatch occurs, a leap will be calculated by selecting the minimum leap among those generated by all patterns using the bad character criterion.

C. J. Coit, S. Staniford, and J. McAlerney proposed the AC_BM algorithm[8], which is very similar to the Setwise BMH algorithm, in 2002. It is different from Setwise BMH as it constructs a tree of all patterns based on the prefixes instead of suffixes. Also, AC_BM takes both bad character and good suffix heuristics to calculate the leap when there is a mismatch.

The focus of this paper is on multi-pattern string matching algorithms for NIMS. Excluding the single-pattern string matching algorithm BM, the other four algorithms can be classified into three categories. Automaton based AC, trie based Setwise BMH and AC_BM, and table based WM. Generally speaking, table structure is likely to be more efficient in space, since they do not need to store every pattern in the data structure. However, it is hard to judge which category would be better in searching time, because the three categories all benefit from their own advantages: automaton based algorithm do not need to do naïve comparisons as they store the matching result at each node; trie based algorithms narrow down the comparison scope by leveraging their compact data structure; while table based algorithms provide very fast search speed through direct table lookups even though they have to perform naïve comparisons when there is a potential match.

The Recursive Shift Indexing (RSI) algorithm proposed in this paper is based on tables, and blocks of two or three characters rather than one character are employed to generate larger potential leap, which is similar to WM. However, the RSI algorithm further engages a heuristic with a combination of the two neighbouring blocks, which increases the probability of getting a leap. Moreover, the potential match table in the third phase keeps a trace of the potential match patterns, which largely decreases the comparison overhead when potential match exists. Design of the bitmaps and recursive tables enhances RSI to do string matching more efficiently. To generate a leap, it only needs two or three table lookups, which is quite efficient in time. Larger leaps can be generated through the recursive structure of tables, compared with Setwise BMH and AC_BM.

3 Recursive Shift Indexing Algorithm

RSI algorithm uses block heuristics to generate leaps, slightly different from that of BM. The idea derives from such an observation: the single bad character heuristic used by BM performs well in single-pattern string matching algorithms, but in multi-pattern cases, the occurrence probability of each character at a position will be higher, and hence the probability to get leaps decreases along the increase number of patterns. Fortunately, multi-character block heuristic has better chance to generate a leap, because the combination of two characters can diminish its probability of occurrence to a great extent. Thus RSI uses two-character block heuristic to eliminate some unnecessary search of the bad characters. The two-character block does not have to be bad characters. Therefore the block heuristic can be applied directly by indexing and the additional comparisons to locate the bad character can be omitted.

The following example illustrates the two-character block heuristic, compared with one bad character heuristic in BM.

One bad character heuristic:

```
Pattern:         AT-THAT
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
                       ↑
Pattern:            AT-THAT
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
```

Two-character block heuristic:

```
Pattern:         AT-THAT
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
                      ↑↑
Pattern:               AT-THAT
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
```

RSI has three preprocessing phases: two Block Leap Tables (BLTs) are generated in the first phase, one Further Leap Table (FLT) in the second phase, and one Potential Match Table (PMT) in the third phase.

3.1 Phase 1: Block Leap Tables

In order to diminish the probability of non-leap resulting from block heuristic, two BLTs are established in the preprocessing phase 1: block#1 stores the leaps based on the first block, namely the first and second rightmost characters, and block#2 stores the leaps based on the second block, namely the third and fourth rightmost characters. The maximum leap of the two leaps generated by the two blocks is applied.

The effectiveness of the two blocks can be illustrated as follows, using the same example.

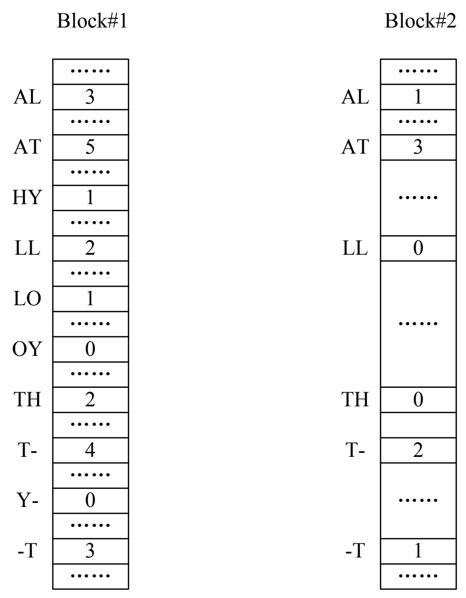
Block#1 of two-character block heuristic:

```
Pattern:         AT-THY-
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
                      ↑↑
Pattern:         AT-THY-
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
```

Block#2 of two-character block heuristic:

```
Pattern:         AT-THY-
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
                    ↑↑
Pattern:             AT-THY-
String:  WHICH-FINALLY-HALTS. AT-THAT-POINT
```

Obviously, the combination of the two block heuristics results in a leap of five, which is much better than the non-leap result of one block heuristic.



	Block#1
	……
AL	3
	……
AT	5
	……
HY	1
	……
LL	2
	……
LO	1
	……
OY	0
	……
TH	2
	……
T-	4
	……
Y-	0
	……
-T	3
	……

	Block#2
	……
AL	1
	……
AT	3
	……
LL	0
	……
TH	0
T-	2
	……
-T	1
	……

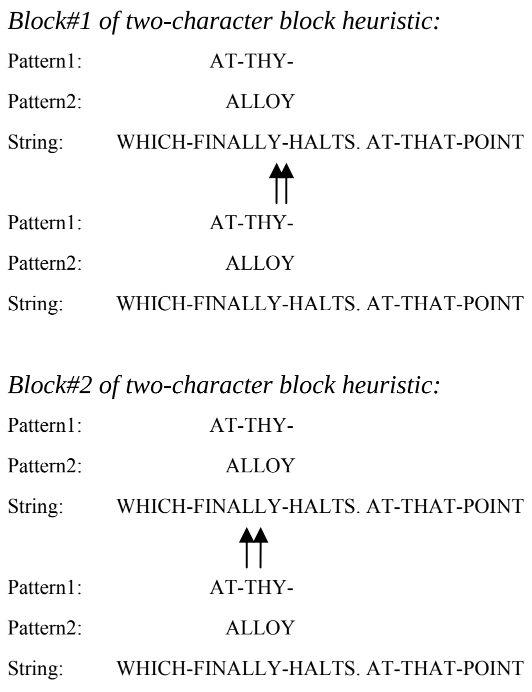
Fig. 1. Example of Block Leap Tables

Two BLTs are created in phase 1 to store the leap values generated by the two blocks, respectively. Each BLT has 2^{16} or 65536 entries, as each block is of two characters (16 bits). In general, block size can be 3 or more, but in the examples and experiments throughout the paper, block size *B* is set the value 2.

For multiple patterns, the leap value in the BLT entries will be the minimum value of all patterns. For instance, in case of two patterns: {AT-THY-, ALLOY}, entries of the BLTs is filled as shown in Fig.1.

3.2 Phase 2: Further Leap Table

Although two BLTs are set up to increase the expectation probability of a long leap, there are circumstances where they may still not provide enough heuristic for better results. Think about the situation that the number of patterns explodes. Large number of patterns debases the expectation value of the leap. With real life patterns and texts, the probability could be larger in worst case. For example, if we have the patterns: {AT-THY-, ALLOY}, and the text contains large quantities of the ordered string "FINALLY-", there will be lots of positions at which the leap values generated by both blocks are zero, which means unnecessary comparisons cannot be avoided here.


Block#1 of two-character block heuristic:
Pattern1: AT-THY-
Pattern2: ALLOY
String: WHICH-FINALLY-HALTS. AT-THAT-POINT
Pattern1: AT-THY-
Pattern2: ALLOY
String: WHICH-FINALLY-HALTS. AT-THAT-POINT
Block#2 of two-character block heuristic:
Pattern1: AT-THY-
Pattern2: ALLOY
String: WHICH-FINALLY-HALTS. AT-THAT-POINT
Pattern1: AT-THY-
Pattern2: ALLOY
String: WHICH-FINALLY-HALTS. AT-THAT-POINT

To further improve the performance of the RSI algorithm, a FLT is introduced in the second phase. This table is an intersection of zero entries of the two BLTs in phase 1 and the synthetical effect is to inspect the text by a four-character block heuristic. The FLT is only queried when the two leap values in phase 1 are both zero.

In phase 1, a temporary bitmap is established for each zero entry of a BLT. Each bit of the bitmap represents a pattern. For example, the bitmap “00100001” indicates that pattern#1 and pattern#6 have a zero leap value. The bitmap linking to a FLT entry is the intersection of the two temporary bitmaps in phase 1. A bitwise logic AND of the two temporary phase 1 bitmaps results in the new one of phase 2. For instance, if the bitmap of block#1 is “00100001” and the bitmap of block#2 is “00101000”, then we get a combined bitmap of “00100000” for the corresponding entry of the FLT, which indicates a potential match only exists for pattern#6, and the entry will be set to zero. Naïve comparison is needed here to inspect whether the potential match of pattern#6 is a real match. But if the bitmap of block#1 is “01010000” and the bitmap of block#2 is “10001000”, we will get a combined bitmap of “00000000”, indicating there is actually no match. Instead of doing an unnecessary comparison here, we can lookup the FLT (using four-character block heuristic) to find a new leap that is not zero.

Since the FLT is an intersection of the two bitmap chunks of phase 1, its size is the product of the sizes of two chunks. The utmost is the square of the number of patterns, because bitmap chunk size (total number of BLT entries with zero leap) in phase 1 cannot be larger than the number of patterns.

3.3 Phase 3: Potential Match Table

The temporary bitmap chunk in phase 2 is set up for creating the PMT. Each bitmap is associated to a zero entry in the FLT. Only at these positions (or for corresponding patterns), there exist potential matches and thus naïve comparisons are needed. In phase 3, the potential match pattern indices are extracted from the bitmaps of phase 2 and then store the indices in an array to accelerate the search. Thus when naïve comparisons are carried out, only those potential match patterns rather than all the patterns are inspected. Once the PMT is filled, the whole preprocessing is accomplished, and all the temporary bitmaps in phase 1 and phase 2 can be released.

3.4 The Lookup Process

Note that we assume all the patterns are longer than four characters here; otherwise the recursive table structure will need to be adjusted. For large number of very short patterns, unnecessary comparisons are hard to avoid, thus AC can be employed as an efficient algorithm. Before the search begins, all the patterns are left aligned and the text is right aligned with the shortest pattern. The searching is doing from right to left. Each of the inspecting process is quite simple:

firstly we extract the two blocks of two-character from the text and lookup the two BLTs for leap values. If the maximum of them is not zero, we can shift the text to the right by this value; otherwise, no leaps are generated by the two-character blocks, and the FLT will be searched to lookup for the four-character block. If the leap value is not zero, we can shift the text to the right accordingly; otherwise, a naïve comparison is carried out based on the heuristics from the PMT.

4 Theoretical and Experimental Analysis

4.1 Theoretical Analysis

We now examine the average performance of the proposed RSI algorithm using a probabilistic model. Following the common practice of some previous work [3], the performance is measured in terms of the number of comparisons that need to be done per character of shift.

$$Performance = \frac{comparisons}{shifts} \qquad (1)$$

Let the size of the alphabet be a, the block size be B, the minimum length of all patterns be m, and the number of patterns be k. We assume uniform distributions of characters in both the text and all k patterns, so the probability of each character Pc will be $1/a$. The probability of *shift* value equaling s is the probability of s consecutive $2B$ character blocks in the text matching none of the patterns ($(1 - Pc^{2B})^{sk}$) and the final $2B$ character block matching ($[1 - (1 - Pc^{2B})^{k}]$):

$$\Pr(shift = s) = \begin{cases} (1-Pc^{2B})^{sk}[1-(1-Pc^{2B})^{k}] & , 1 \le s \le m-2B \\ (1-Pc^{2B})^{(m-2B)k}[1-(1-Pc^{m-s})^{k}] \prod_{i=1}^{s-m+2B} (1-Pc^{2B-i+1})^{k} & , m-2B+1 \le s \le m \end{cases} \qquad (2)$$

Note that when $m - 2B + 1 \le s \le m$, the probability equation is a little complicated, because at the edge of comparison window, blocks of less than $2B$ characters are needed. Thus the expected value of *shift* will be:

$$E[shift] = \sum_{s=1}^{m} s * \Pr(shift = s) \qquad (3)$$

Now consider the expectation of comparisons. Since we inspect two characters as a block in phase 1, only two comparisons are needed. If it is necessary to go to phase 2, one extra comparison is added, resulting in three comparisons. These two probabilities can be calculated as follows:

$$\begin{aligned}\Pr(comparison = 2) &= 1 - [1 - (1 - Pc^{B})^{k}]^{2} \\ \Pr(comparison = 3) &\doteq [1 - (1 - Pc^{B})^{k}]^{2}\end{aligned} \tag{4}$$

The probability to go to phase 3 is nearly $\Pr(comparison = 3) / pNum$, where $pNum$ is the number of patterns). If we designate $aveLength$ as the average length of all patterns, the contribution of phase 3 to the expectation value of *comparison* is at most $pNum * aveLength * \Pr(comparison = 3) / pNum = aveLength * \Pr(comparison = 3)$. Thus the approximate expectation value of *comparison* is:

$$E[comparison] = 2 * \Pr(comparison = 2) + (3 + aveLength) * \Pr(comparison = 3) \tag{5}$$

If alphabet size a is 256 (2^8 for 8 bit characters), the block size B is 2, the patterns are all of the length 16, and the number of patterns is 500, the expectation value of *shift* will be 15.1325, and the expectation value of *comparison* will be 2.0010. This means that if the text length is n, only $0.1322\,n$ times of comparisons are needed.

Besides time performance, space occupation is another crucial factor of the string matching algorithms, since many hardware platforms have their own limits in memory. Thus algorithms requiring less memory are likely to achieve their ideal performance, while algorithms requiring too much memory may encounter a slow down due to the space limitation.

RSI algorithm has a relatively stable data structure, so we can evaluate the space performance simply according to the characteristics of the patterns, including pattern number and pattern lengths. Simply, we can assume a uniform length for all the patterns and use $pLen$ to designate it. Still $pNum$ stands for the number of patterns. As illustrated in Section 3, in phase 1 the two BLTs of the size a^{B} are required; in phase 2, the maximum size of the FLT is $pNum * pNum$; and in phase 3, the utmost size of the PMT is $pNum * MAX\_P\_SET$, where MAX_P_SET is the maximum number of patterns that might match simultaneously at one time. Assume that each entry is 2 bytes in size, the total memory needed for RSI is:

$$TotalMemory = 2 * (2 * a^{B} + pNum * pNum + pNum * MAX\_P\_SET) \tag{6}$$

If alphabet size a is 256, the block size B is 2, the number of patterns is 500, and MAX_P_SET is 10, we get a total memory of 772 KB. Even with the number of patterns being

1000, the utmost total memory will be no more than 1141 KB. In addition, the memory occupation has nothing to do with the pattern length *pLen* .

4.2 Experimental Results

Experiments are designed to verify the performance of the proposed algorithm both on searching time and space occupation, and to compare it with other algorithms. For the purpose of a fair comparative analysis, Setwise BMH is implemented, and pure C source code of AC, AC_BM and WM are extracted from Snort version 2.3.3.

Tests are performed on a PC with AMD Athlon 1.140G CPU, 64KB L1 Cache and 384MB memory. Comparisons are done from two aspects that compliment each other: one with fixed number of patterns and varying pattern lengths; the other with fixed pattern length and varying pattern numbers. All patterns are generated randomly with equal length, but the text is self-correlated, which means part of the text is generated randomly but the rest is generated according to that part. For instance, if we need a text of the length 10,000, we generate the first 500 characters randomly, and the rest of the text are generated like this: each time we pick out several characters from the first 500 characters and append them to the end of the text until the length arrives at 10,000; the position and the length of each pick are both random. The aim to derive such text is to guarantee that there exist a number of matches, as we know random patterns and text will result in few matches. Thus we can better simulate the patterns and traffic in real network. Though these are only meaningless characters, they provide a meaningful reference of the performance.

Fig.2 shows the total searching time for a text with 50,000 characters and 500 patterns with a varying length. The time in Fig.2 is the accumulative time of 1,000 repeated times of lookup. It can be seen that RSI bears a good time performance superior to the other algorithms. Meanwhile, space occupations detailed in Fig.3 show that RSI need almost as less memory as WM, regardless of the pattern length, while AC, AC_BM and SBHM need more memory and the memory size is proportional to the pattern length.

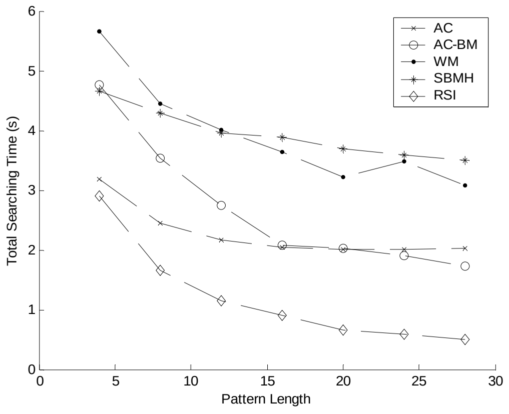


Fig. 2. Time with fixed pattern number

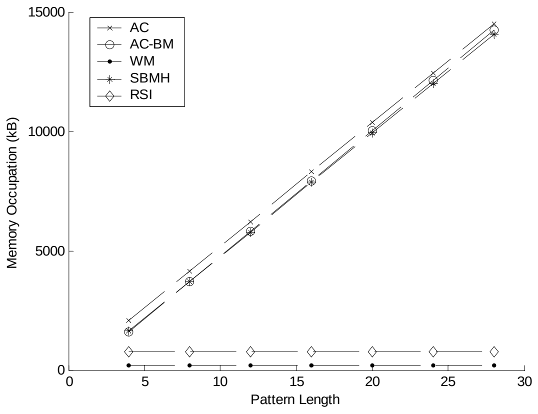


Fig. 3. Space with fixed pattern number

Fig.4 and Fig.5 are the results with fixed pattern length of 8 and a varying pattern number from 10 to 1000. From Fig.4 we can see that RSI consumes much less time compared with the other algorithms and the total searching time does not increase much as the pattern number rises. Fig.5 illustrates the space performance of the algorithms, which reveal that memory requirements of all the five algorithms go up as the number of pattern grows. However, RSI algorithm does not need so much memory as many of the others, and WM needs the least memory. Tests with other fixed pattern length such as 16 and 24 have been carried out, and the result curves are similar.

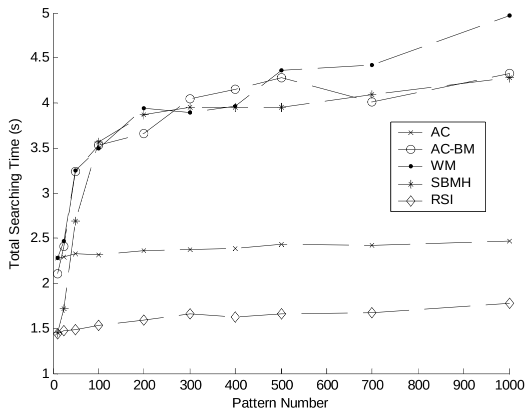


Fig. 4. Time with fixed pattern length 8

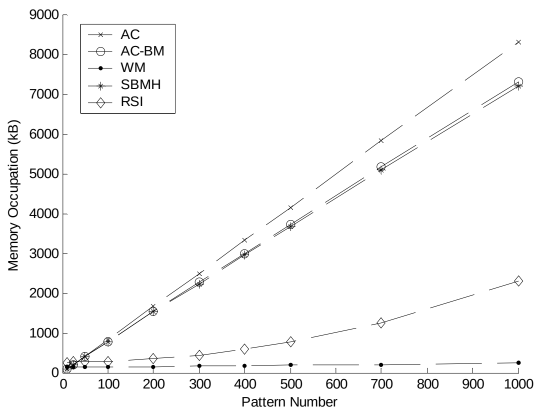
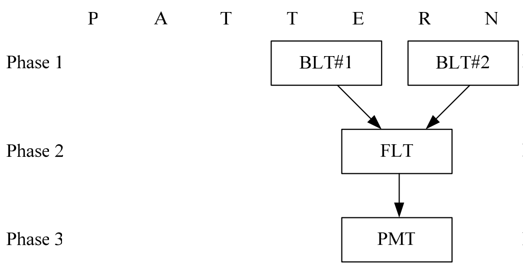


Fig. 5. Space with fixed pattern length 8

From the experimental results, it can be concluded that the Recursive Shift Indexing algorithm possesses outstanding performances both in time and space, consistent with the theoretical analysis. The only shortcoming might be the preprocessing time, which is a bit long when the number of patterns is large and the length of patterns is long. With 1000 patterns of the length 20, 3 to 4 minutes is required in the test. Nevertheless, preprocessing time is not as crucial as searching time to NIMS, because the update of pattern libraries does not happen very frequently.

5 Variations

Several variations and improvements of RSI are possible. Throughout the paper, we adopt the three-phase table structure illustrated in Fig.10. However, RSI can be extended to engage more blocks during one inspection, and varying the block size will bring changes in time and space complexity. Note that in our examples, we use two blocks with block size $B=2$. Fig.6 illustrates the recursive table structure in this paper. Attention should be paid that Fig.7, Fig.8, and Fig.9 are within the possible variations. There could be even more variations.



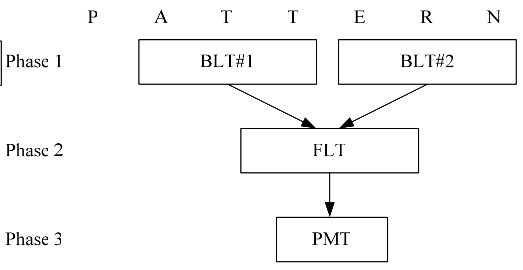
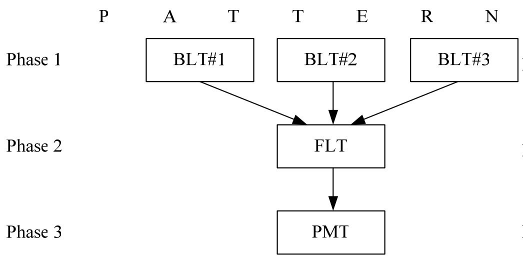


Fig. 6. RSI table structure example **Fig. 7.** RSI table structure variation 1



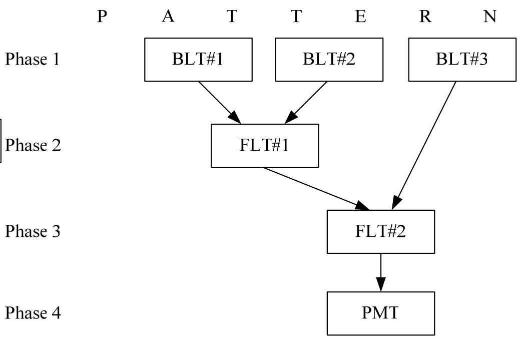


Fig. 8. RSI table structure variation 2 **Fig. 9.** RSI table structure variation 3

Following the theoretical analysis in Section 4, we can perform time and space evaluations for each of the variations respectively. Take variation 1 for example. For comparison convenience, here we still let alphabet size a be 256, the pattern length 16, and the number of patterns 500, but the block size B is set to 3. We can calculate the expectation value of *shift* by formula (2) and formula (3), and the result is 15.1326. Formula (4) and formula (5) are used for evaluating the expectation value of *comparison* and the result turns out to be 2.0000. By means of formula (1),

$0.1322n$ times of comparisons are required if the text length is n. The utmost space occupation can be evaluated through formula (6) and the value is 67,619 KB.

Similarly we can evaluate the time and space needed according to the table structures in Fig.8 and Fig.9. Table 1 is a complete comparison of the four kinds of structure.

Table 1. Time and Space performance comparisons between example and variations

Structure	Time (*shifting rate)*	Space (*KB*)
Example	0.1322	772
Variation 1	0.1322	67,619
Variation 2	0.1982	250,403
Variation 3	0.1982	1,403

From Table 1 we can tell that theoretically the example structure is expected to perform well both in time and space. The limitation is that all patterns should be longer than four characters. In structure variations from 1 to 3, all patterns need to be longer than six characters. However, the evaluation above is merely based on an assumption of random characters in patterns and text. In real applications, structure variation 3 might perform a higher efficiency if the network traffic contains many strings that share large similarities with the patterns. The choice between the example and variations depends on the actual circumstances.

6 Conclusions

Multi-pattern string matching algorithms are still formidable problems in network intrusion management systems nowadays. This paper proposed a novel high performance algorithm based on both theoretical analysis of a probability model and experimental comparisons with synthesized data. The algorithm is demonstrated to be superior to the other algorithms, with balanced time and space performance. The overall performance is attractive and rather stable, regardless of the pattern length.

Future work can be conducted on hardware implementation of the algorithm. Due to the concise structure, it should be considerably easy to implement RSI on FPGA or ASIC. The authors are considering embedding the algorithm into a realistic NIMS such as Snort. Experiments are now based on pure C source code on Windows XP platform. More convincing results can be provided if the algorithm is tested in a real world network intrusion management system. With realistic patterns indicating practical threats and actual packet payload in network, the average performance of the algorithm can be evaluated, including the online speed of packet processing throughput. The authors are also implementing the algorithm into Intel’s IXP2800 network processor (NPU). RSI adopted to the parallel processing environment can take advantage of the multi-core hardware platform, and it is expected to produce multi-Gbps throughput.

References

1. H. Debar, M. Dacier, and A. Wespi, Towards a taxonomy of intrusion-detection systems, Computer Networks, vol. 31, pp. 805-822, 1999.
2. M. Roesch, Snort: Lightweight intrusion detection for networks, Proc. of the 1999 USENIX LISA Systems Administration Conference, 1999.
3. M. Fisk and G. Varghese, An analysis of fast string matching applied to content-based forwarding and intrusion detection, Technical Report CS2001-0670, University of California – San Diego, 2002.
4. K. G. Anagnostakis, S. Antonatos, E. P. Markatos, and M. Polychronakis, E^2xB: A domain-specific string matching algorithm for intrusion detection, Proc. of IFIP International Information Security Conference (SEC'03), 2003.
5. R. Boyer and J. Moore, A fast string searching algorithm, Commun. ACM, vol. 20, no. 10, pp. 762-772, 1977.
6. A. Aho and M. Corasick, Fast pattern matching: an aid to bibliographic search, Commun. ACM, vol. 18, no. 6, pp. 333-340, 1975.
7. S. Wu and U. Manber, A fast algorithm for multi-pattern searching, Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.
8. 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 (DISCEX II), 2002.

Appendix

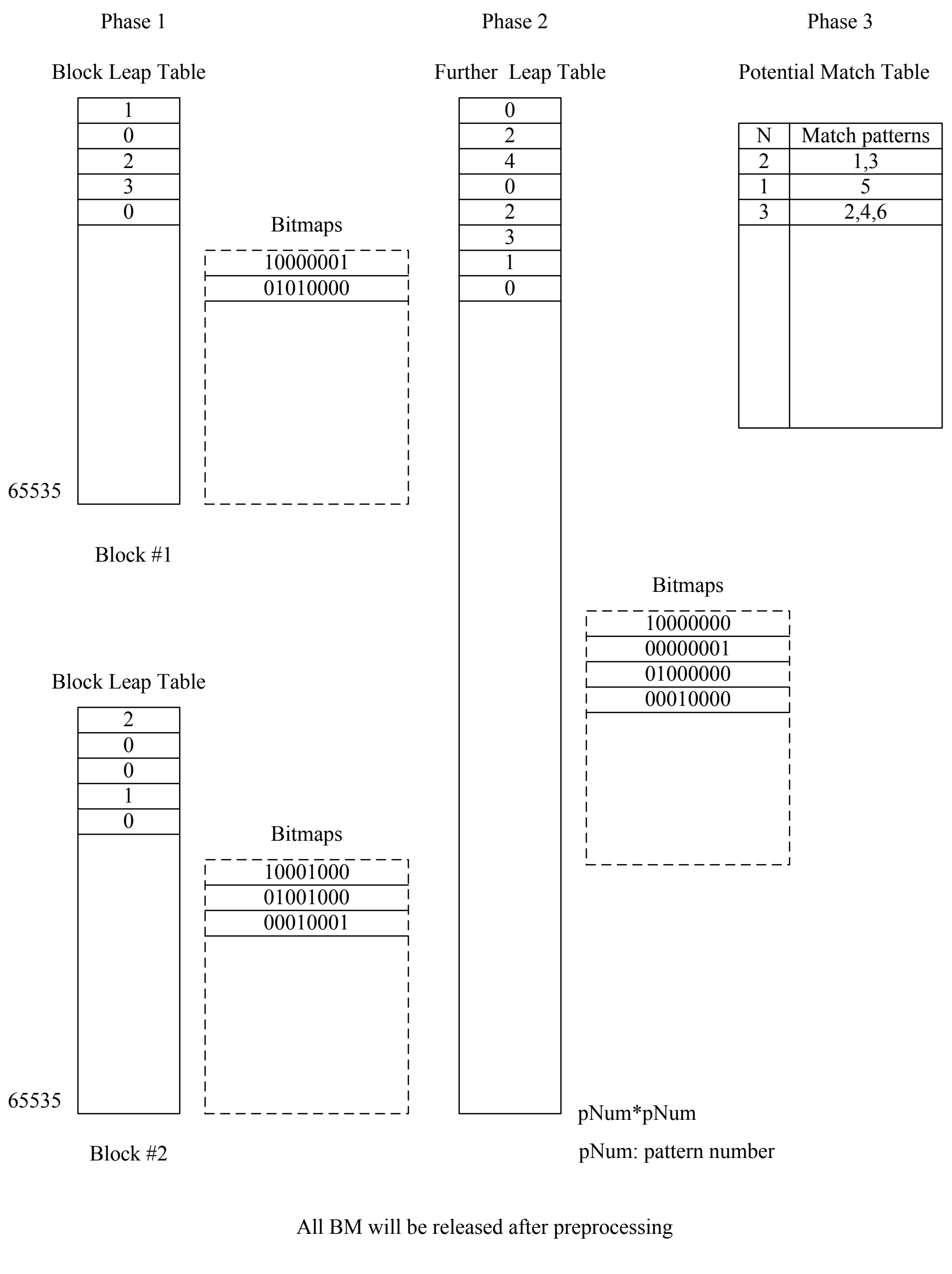


Fig. 10. Full RSI Table Structure