Performing Packet Content Inspection by Longest Prefix Matching Technology

Nen-Fu Huang, Yen-Ming Chu
Department of Computer Science and
Institute of Communication Engineering,
National Tsing Hua University,
HsinChu, Taiwan

Yen-Min Wu, Chia-Wen Ho
Department of Computer Science,
National Tsing Hua University,
HsinChu, Taiwan

Abstract—**This article presents a novel mechanism to perform packet content inspection by longest prefix matching (LPM) technology. It is done by transforming the automaton-based state table lookup problem into the famous LPM table lookup problem. Two key features, symbol-wise prefix and magic state are observed on the state table to make it possible to utilize IP lookup techniques for string matching. The proposed mechanism is verified to be effective through Lulea algorithm. Also, the practicability is evaluated by employing realistic attack signatures and traffic traces. The experimental results indicate that a state table constructed from the Snort 2.4 patterns can be converted into a prefix table that requires only 2.5% of the memory utilized in the original state table. Compared with the state-of-the-art researches, the proposed scheme has more than 3 times of efficiency, achieving a better balance between required memory size and throughput rate.**

***Index Terms*—Packet Content Inspection, Longest prefix matching, String matching.**

I. INTRODUCTION

Deep packet content inspection is one of the most significant challenges and important issues to provide security service in high speed network environment. To achieve high-speed packet content inspection, many string matching algorithms have been proposed. For example, the Boyer-Moore algorithm has the best search performance among current well-known single pattern matching algorithms [1]. When it comes to multiple-pattern matching, Aho-Corasick (AC) algorithm and Wu-Manber (WM) algorithm, which were introduced in 1975 and 1992, respectively, are the two most well-known algorithms [2][3]. AC is an automaton-based algorithm and WM exploits the feature of bad character heuristic.

In network appliances like switches, the most common application of string matching is the layer-2 address resolution. Since the pattern lengths in the layer-2 address lookup are the same (48/64 bits), binary searching, hashing, and CAM/TCAM are the common techniques to apply [4]. Although various researches focus on performing a single pattern search in parallel, they are not suitable for modern network environments [5][6]. This is because the deep packet inspection requires multiple-pattern matching with variable length patterns. For instance, the rules in Snort are of variable lengths [7].

For the demand of performing layer-7 packet content inspection on network appliances, various studies have been proposed and implemented—search filter [8][9], reconfigurable silicon hardware [10], and TCAM-based method [11]. In this paper, a novel scheme, "*String Matching as Longest Prefix Matching*" (*SM as LPM*), is presented for applying the existing layer-3 LPM techniques of IP address lookup to the layer-7 packet content inspection. This paper also explores a new development aspect to LPM, which is regarded as a mature research field today. The LPM algorithm is used to find the longest prefix-matched entry in router's forwarding table. Since the speed of IP lookup is crucial to routers' performance, numerous LPM algorithms have been proposed [12]-[14]. In this paper, we show that the layer-3 IP lookup techniques can be applied to the layer-7 content inspection by employing the analogy between IP lookup and automaton-based string-matching algorithms.

The rest of this paper is organized as follows. In section II, we present earlier researches on automaton-based string-matching algorithms and related studies. In section III, the basic concepts of *symbol-wise prefix* and *magic state* are introduced. Also, the well-known LPM algorithm, Lulea, is applied here as an example to verify the correctness of the proposed scheme. In section IV, the effectiveness of the proposed mechanism is analyzed and evaluated. Finally, some conclusions and future work are given in section V.

II. RELATED WORK

Before introducing the idea of transforming the string matching problem to the LPM problem, let us first review an important model in computer science: finite state automaton, FSA (i.e., finite state machine, FSM). FSA is composed of a finite number of states. Each state can transit to zero or more states depending on the input information. Finite state automaton can be classified into two categories: deterministic (DFA) and nondeterministic (NFA). Generally, DFA has better performance than NFA but it requires more states (memory space). It is worth mentioning that the Donnelly-proposed "*IP Route Lookups as String Matching*" uses finite state automata to

This work was supported by MOE Program for Promoting Academic Excellent of Universities (II) under the grant number NSC-94-2752-E-007-002-PAE, and NSC project under the grant number NSC-95-2221-E007-054.

solve the problem of LPM [15].

A. Automaton-based Algorithm

Although the WM algorithm is superior to all other multiple-pattern matching algorithms in the average case, it only shifts one byte at a time in the worst case. Malicious elements (like hackers) can exploit this feature to attack systems. A research [16] employing a synthetic input string shows that the performance of such algorithms are considerably suffered from algorithm attacks, which can be even declined to that of single pattern string matching. Therefore, general network appliances, especially those related to network security, usually avoid adopting WM algorithm since the device may become a victim of denial of service attack.

In order to avoid these problems, the well-known AC algorithm becomes a popular choice. It constructs an FSA based on the given keyword patterns and performs string matching by state transition. In the worst case, it still has a deterministic processing rate. In AC-based automata, each state requires 256 next state pointers, a pattern pointer, and a corresponding pattern ID (*PID*). Besides, the NFA structure needs to maintain another failure pointer to record the failure path. To reduce the huge amount of memory space requirement, Norton [17] used special compression methods to decrease the memory space when implementing Snort rules on AC; this is the most popular AC implementation nowadays. However, it also lowers the search performance. Besides, Nishimura proposed a speed-up method for the AC algorithm by rearranging the states [18].

N. Tuck, et al modified the NFA-based AC algorithm and used bitmaps that correspond to symbols to record the state transition of the non-failure path [16]. In this way, every node in the finite automaton only uses a pointer pointing to the next state list instead of allocating all the pointers to the next state.

Recently, J.V.Lunteren, et al [19] presented a BFSM-based pattern-matching (BFPM) scheme based on a BaRT routing table search algorithm. BFSM is based on state transition rules that include conditions for the current state and input symbols, which are assigned priorities. However, this scheme is not suitable for software implementation due to its multiple FSM data structure.

III. String Matching as Longest Prefix Matching

The proposed model for performing string matching by longest prefix matching consists of two stages. The first stage performs state transition by LPM-based table lookup. The second stage searches the pattern ID if the output of the first stage is an accepting state.

Fig. 1 explains the execution of the first stage, state table lookup stage. Since the next state is generated from the current state ψ and the input symbol t_r, $Index = \{t_r{:}\psi\}$. Because all symbols used in this paper are ASCII codes, the bit size u of a symbol is 8. The bit size v of a state is decided by the number of whole characters of all patterns and is generally equals to 16 or 32. For example, the total amount of states constructed from Snort 2.4 patterns is 21595 and hence the bit size v of a state is 16. The bit size w of the index is then given by $w = u + v = 8 + 16 = 24$ bits. The *index* is generated by a conjunction of the input symbol that enters the FSM and the current state. It is used to lookup the corresponding next state in state table. The major job of the second stage, pattern search, is to determine whether the output state of state table lookup is an accepting state or not. If it is, then find the matched pattern ID.

We focus on state table lookup here, as it is the key operation to the automaton-based algorithm. In fact, the fundamentality of all kinds of FSA operations is based on state transitions. The state table in table lookup stage can be represented as a state transition matrix depicted in Fig. 2. The variables u and v represent the bit sizes of the symbol and state, respectively. The element $e_{(x,y)}$ represents the appropriate next state when the current state y receives the input symbol x, and all the state transitions in the FSM can be recorded in the matrix. In the remaining sections of this paper, the FSA will be represented in the form of matrix M. This study concentrates on problems involving M for the DFA. We observed two interesting key features: *symbol-wise prefix* and *magic state*, which are described as follows.

A. Symbol-wise Prefix

Among the IP lookup algorithms, an algorithm named DIR-24-8-BASIC was proposed by P. Gupta [13]. In simple words, the algorithm segments the 32-bit IP address *a.b.c.d* into two parts: the 24-bit part (*a.b.c)* and the 8-bit part (*d*). By a straightforward method, the first 24 bits of the IP address is employed to function as the index for 2^{24} entries. The entry content of a prefix whose length equals to or less than 24 bits is the next hop, and for prefixes with length greater than 24 bits, the next hop will be found in the level-2 table whose index comprises the last 8 bits of the IP address. The concept of this IP lookup algorithm can be adopted straightforwardly to use 24 bits as the index in our proposal. This makes both the proposed algorithm and that in [13] have almost the same table structure and operations. The original 2^{24} elements in the automaton matrix can be rearranged as shown in Fig. 3.

Conventionally, the next state is decided by the current state and the input symbol in the FSM. In the scheme shown in Fig. 1, we straightforwardly assume the combination of the current state and input symbol to be the index and determine the next state using direct-lookup mechanism. The next state and *Index* are seen as the next hop and IP address (most significant 24-bit portion) in the LPM, respectively.

Now, let us have some critical definitions: Let $Index_{(x,y)} = \{w_{23}, w_{22}, \ldots, w_{16}, w_{15}, \ldots, w_1, w_0\}$, where w_{23} represents the 24th bit, which is the most significant bit (MSB), and w_0 represents

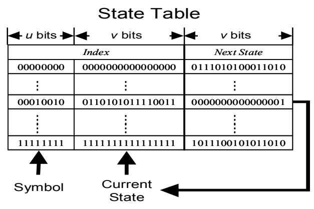

Fig. 1. Illustration of state table lookup.

the first bit which is the least significant bit (LSB). The $Index_{(x,y)}$ is generated by the combination of input string symbol x (8 bits) and current state y (16 bits). There are two ways to design the data structure of the state table: *state-wise prefix* or *symbol-wise prefix*. For the state-wise design, the current state and input symbol are placed in the MSB-side and LSB-side, respectively, and we have $Index_{(x,y)} = \{v_{15},\ldots, v_1, v_0, u_7, u_6,\ldots, u_0\}$. This design is suitable for the case when M is an NFA.

Nevertheless, it is interesting to see that when M is a DFA, for each symbol x, most of $e_{(x, y)}$ have the same value (next state) for different current state y. For example, Fig. 4 shows the partial DFA table for Snort 2.4 constructed by the AC algorithm. We can see that most of the next states for input symbols 0x47, and 0x4C are 269, and 527, respectively. This generates the idea of *symbol-wise prefix* where the input symbol and current state are placed on the MSB-side and LSB-side, respectively, so that $Index_{(x,y)} = \{ u_7, u_6,\ldots, u_0, v_{15},\ldots, v_1, v_0\}$.

Based on this symbol-wise prefix data structure, it is observed that for the next state array with 2^{24} entries, many continuous entries have the same next state number. In other words, it is very common to observe the relation $e_{(x,y-\alpha)}=\ldots=e_{(x,y)} = e_{(x,y+\beta)}$, where $0 \le \alpha, \beta \le 2^v-1$. This provides the opportunity to aggregate many consecutive entries into one entry with shorter prefix length. For the sake of simplicity, consider the example of $u = 2$, $v = 3$, and for an input symbol x, $e_{(x, y-1)} = e_{(x, y)}$. In this case, the state table is original of 32 entries ($2^{u+v}= 2^5=32$). Assume that $x = 3(11)$, $y = 5(101)$. Then we have $e_{(11, 100)} = e_{(11, 101)}$. In other words, the 28th (11100) entry and 29th (11101) entry of the state table can be merged into one entry with prefix 1110*. The star symbol "*" represents "don't care" similar to its meaning in routing prefixes. Consequently, the prefix lengths of $Index_{(11, 100)}$ and $Index_{(11, 101)}$ are reduced from 5 bits to 4 bits. Based on this concept, if there are four consecutive entries of state table with the same entry value (the same next state), then these four entries can be merged into one entry with prefix "…**", and so on. This *symbol-wise prefix* is the primary key that enables the state table lookup to be processed by using the LPM lookup algorithm.

Fortunately, as we have figured out in Fig. 4, for each input symbol, most of its next states are with the same value, and this provides a good chance to merge most of the entries of state table so that the total number of entries can be dramatically reduced. For example, when $u = 8$ and $v = 16$, for the matrix M constructed by the Snort 2.4 patterns, the number of entries is dramatically reduced to 590,453 from 16-million (2^{8+16}) entries when the symbol-wise prefix approach is employed, requiring only 3.5% of the original number of entries. Fig. 5 shows the distribution of the symbol-wise prefix, and the shortest prefix length can be only 10 bits.

$$M \equiv \begin{bmatrix} e_{(0,0)} & e_{(1,0)} & \cdots & \cdots & \cdots & e_{(2^u-2,0)} & e_{(2^u-1,0)} \\ e_{(0,1)} & \ddots & \ddots & \ddots & \ddots & \ddots & e_{(2^u-1,1)} \\ \vdots & \ddots & \ddots & e_{(x,y-1)} & \ddots & \ddots & \vdots \\ \vdots & \ddots & e_{(x-1,y)} & e_{(x,y)} & e_{(x+1,y)} & \ddots & \vdots \\ \vdots & \ddots & \ddots & e_{(x,y+1)} & \ddots & \ddots & \vdots \\ e_{(0,2^v-2)} & \ddots & \ddots & \ddots & \ddots & \ddots & e_{(2^u-1,2^v-2)} \\ e_{(0,2^v-1)} & e_{(1,2^v-1)} & \cdots & \cdots & \cdots & e_{(2^u-2,2^v-1)} & e_{(2^u-1,2^v-1)} \end{bmatrix}$$

Fig. 2. Automaton Matrix.

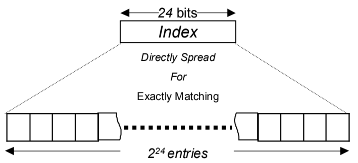

Fig. 3. Direct-lookup mechanism.

B. Magic State

In the previous subsection, we proposed the symbol-wise prefix that establishes a correspondence between the DFA state table and routing table. This successfully solves the DFA suitable problem mentioned in [16]. As we have pointed out before, the DFA constructed by the AC algorithm provides an interesting feature that for each input symbol x, most of the current states have the same next state value. We define this next state as the "*magic state*" of x (denoted as $ms(x)$). Consider the case shown in Fig. 4, the magic states for symbols 0x47 to 0x4C are 269, 139, 151, 803, 1061, and 527, respectively.

The magic state feature can be further applied to the symbol-wise prefix to reduce the prefix length. The construction procedure for the symbol-wise prefix can now be slightly modified as: In matrix M, *if* $e_{(x,y)} = ms(x)$, the corresponding $Index_{(x,y)}$ will be transformed to $Index'_{(x,y)} = \{w'_{23}, w'_{22},\ldots, w'_{16}, w'_{15},\ldots, w'_1, w'_0\} = \{w_{23}, w_{22},\ldots, w_{16}, *, *,\ldots, *, *\}$. Thus, if $e_{(x, y)}$ is a magic state, the values from w'_{15} to w'_0 of the corresponding $Index'_{(x,y)}$ will be "don't care". More precisely, all the entries with the next state equals to the magic state $ms(x)$ can be merged into one symbol-wise prefix with a length of 8. To prove the effectiveness of the magic state concept, an experiment with the signatures of Snort 2.4 is conducted. Fig. 6 shows the correctness of our proposition. There are 256 symbol-wise prefixes (corresponding to the magic states for each symbol) that are 8-bit long, and the total number of prefix entries decreases to 273,212. In other words, with the magic state, the number of symbol-wise prefixes is decreased by approximately 50% and the total number of the lookup table entries is dramatically reduced to only 1.5% (273k/16-million).

Besides the fact that *the magic state* can decrease the number of symbol-wise prefixes, it has a much more important meaning for performing string matching by longest prefix matching. Compare the prefix distributions shown in Fig. 5 and Fig. 6, we can find that the magic state feature not only decreases the number of prefixes considerably but also concentrates the prefix distributions. In other words, after the magic state feature is applied, the distribution of the symbol-wise prefix will appear only in five lengths: 8, 21, 22, 23, and 24. A distribution like this will make the non-trie LPM algorithms, like hash-based and

		Input Symbol(ASCII)					
		0x47	0x48	0x49	0x4A	0x4B	0x4C
Current State	16	269	1037	1016	803	984	527
	17	269	139	151	803	1061	527
	18	12551	14000	12435	19727	1061	13877
	19	3063	12373	4190	803	1061	14843
	20	269	139	151	803	1061	527
	21	269	139	19	803	2558	527
	22	269	3464	151	803	1061	527
	23	269	4145	4190	803	1061	257
	24	269	139	15288	803	1061	527
	25	269	139	151	803	1061	527

Fig. 4. Partial DFA table of Snort 2.4.

TCAM-based algorithms, have excellent performance [20]. For example, consider the approach proposed by Lim [21]; it divides a prefix into groups based on the prefix length and sends each group of prefixes to the corresponding engine. Since the prefixes are grouped based on a maximum of five lengths, we need only five engines to process the lookup operation.

C. Demonstrations

From the experiments in Fig. 5 and Fig. 6, we can see that the symbol-wise prefix is similar to the IP routing prefix and can effectively decrease the index requirement. We have straightforwardly used direct-lookup mechanism for string matching, and next, we would like to further prove that dealing with the symbol-wise prefix is the same as dealing with the IP prefix through applying a routing lookup algorithm.

M. Degermark presented a data structure for forwarding table named Lulea algorithm designed for fast routing lookups [12]. By an ingenious design, it enables the IP routing table to be compressed for storing in the cache (500~600 KB for 40,000 entries) of a processor through three elegant structures: *code word array*, *base index array,* and *maptable.* The algorithm got conceit and spawned an industry of follow-up articles [22]. Therefore, it is employed as the longest prefix matching algorithm without changing its algorithm and data structure.

In Lulea [12], the routing prefix is segmented into three levels—one 16-bit level and two 8-bit levels. The data structure is mainly placed in level-1. Reviewing the discussion on the symbol-wise prefix, concerning Snort 2.4, there were 256 symbols and no more than 65,536 total states and the longest prefix was 24-bit long. If we segment the symbol-wise prefix into two levels of 8 bits and 16 bits, we can find its advantageous like the case employing Huang's NHA [14]. This means that the second level of the symbol-wise prefix is exactly suitable for the design of Lulea's level-1. Therefore, we have to maintain 256 pairs of code word and base index arrays but just one *maptable* data structure.

When processing the next state lookup, we use the first 8 bits (w_{23}~w_{16}) of *Index* to decide which code word array and base index array are needed for the lookup. We then execute level-1 of the Lulea algorithm with the last 16 bits (w_{15}~w_0) to obtain the next state value. Considering Fig. 7 as an illustration, the first 8 bits representing the symbol that decides which code word array and base index array are used; these arrays are represented by hollow arrows in the figure. Next, we perform a lookup by employing the Lulea algorithm; this is represented in the dotted box on the right-hand side in the figure. By using the symbol-wise prefix, the string-matching problem can be successfully handled by the LPM algorithm.

IV. Evaluation and Analysis

To evaluate the effectiveness of the proposed analogical relationship as well as the transformation procedure between string matching and LPM, the Snort 2.4 is taken as the patterns and the Defcon9 traces [23] are used as the input strings.

Table I is the evaluation result for four implementations from the viewpoint of memory size E_m (MB), software throughput rate E_t (Mbps), and overall efficiency E. The efficiency value is defined as $E = E_t/E_m$, which represents throughput rate generated by each unit of memory. AC-DFA and AC-NFA represent the DFA and NFA in the traditional AC algorithm, respectively. AC-Bitmap is the algorithm for the NFA with bitmap that was proposed in [16]; while AC-Lulea refers to the implementation of the LPM algorithm [12] previous introduced. In Snort 2.4, the number of signatures (keyword patterns) is around 2390 with a total of 35K characters. Through Norton's AC implementation, the content is transferred to a DFA of 21,595 states and among them, 8,477 states are accepting states. The required memory space is approximately 56 MB according to the studies that have analyzed the AC algorithm [16][17]. In addition, by a comparison with several results of traces in various networks, we can find that the number of state transitions in a DFA is 30% less than that in an NFA.

When Lulea algorithm is applied, the symbol-wise prefix is taken into consideration. Since the original 21595 × 256 indexes are reduced to 590,453 symbol-wise prefixes (also shown in Fig. 5), the required memory space is decreased considerably. The memory space for Lulea algorithm is reduced to less than 1.4 MB. Lulea algorithm is also implemented on a Windows XP platform to verify the correctness of the proposed mechanism. The software throughput for this implementation is also shown in TABLE I. Although the throughput for Lulea algorithm is not as good as traditional AC implementations because the proposed model involves the processing of the data structure (like ***popcount***), the performance is still three times better than that of the compressed NFA with bitmap in [16]. Comparatively speaking, the AC-Lulea algorithm is much better than the AC-Bitmap algorithm.

Based on TABLE I, we observed that there is a trade-off between the memory size requirement and the throughput rate. To make the comparison more practical and realistic, we take both throughput and memory usage into consideration, that is, overall efficiency (E). The higher E value represents better balance between performance and memory usage. Among all implementations listed in TABLE I, AC-Lulea has the best efficiency, about 7 times of that of AC-NFA, 3 times of that of AC-Bitmap, and more than 2 times of that of AC-DFA. The statistics indicate that through consumption of each memory unit, AC-Lulea could achieve the highest throughput rate. Since

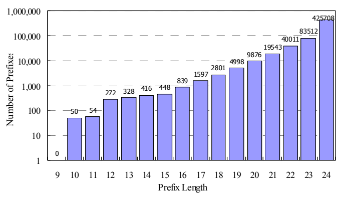

Fig. 5. Length distribution of symbol-wise prefix in DFA.

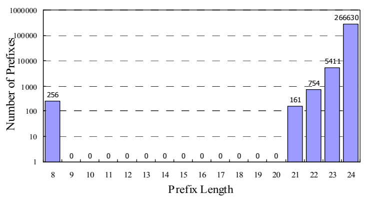

Fig. 6. Length distribution of symbol-wise prefix with 256 magic states.

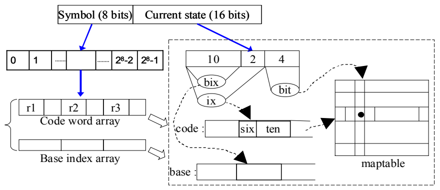

Fig. 7. Next state lookup using Lulea algorithm.

the proposed implementation requires only a tiny amount of memory, it provides high flexibility in various environments, especially in resource-limited network appliances.

V. Conclusions

A novel mechanism to perform string matching by longest prefix matching (LPM) has been proposed in this paper. We focus on converting the state table lookup problem in automaton-based string matching algorithms to the famous IP routing table lookup problems. Two characteristics, *symbol-wise prefix* and *magic state* are observed to make it possible to utilize IP LPM lookup techniques for string matching. By applying the two key features, a state table established from the Snort 2.4 patterns is successfully transferred into a prefix table that requires only 2.5% of the memory utilized in the original state table. Compared with past NFA-based string matching researches, the proposed scheme achieves more than 3 times of efficiency.

By demonstrating that the string matching can be performed by LPM, it opens a window for designing cost-effective security switches/routers based on the commodity L3 switches or routers. As many modern L3 switches or routers employed the LPM algorithms for IP address lookup, we now have the opportunity to upgrade these devices to layer-7 with deep content inspection capability based on their original hardware platforms. Particularly, for speed consideration, it is even better if TCAM is designed in the hardware platform as several efficient TCAM-based LPM algorithms can be applied.

TABLE I. Efficiency of different algorithms

	E_m (MB)	E_l (Mbps)	Efficiency E
AC-DFA	56	288	5.14
AC-NFA	56.2	104.1	1.85
AC-Bitmap	1.4	5.2	3.71
AC-Lulea (LPM)	1.4	17.2	12.29

References

[1] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, Session 10, Oct. 1977, pp. 761–772.
[2] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Communications of the ACM*, vol. 18, issue 6, Jun. 1975, pp. 333–340.
[3] 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
[4] R. Seifert, *The switch book: a complete guide to LAN switching technology*, John Wiley & Sons, 2000
[5] A. N. M. E. Rafiq, M. W. El-Kharashi, and F. Gebali, “A Fast String Search Algorithm for Deep Packet Classification,” *Computer Comm.*, vol. 27, no. 15, pp. 1524–1538, Sept. 2004
[6] F. Gebali and A. N. M. E. Rafiq, “Processor Array Architectures for Deep Packet Classification,” *IEEE Transactions. Parallel and Distributed Systems*, vol. 17, issue. 3, pp. 241–252, March 2006.
[7] M. Roesch, “Snort: Lightweight intrusion detection for networks,” *USENIX 1999 LISA Systems Administration Conference*, November 1999. Available: http://www.snort.org/
[8] N. F. Huang, Y. M. Chu, J. L. Chen, and K. J. Huang, “A non-Computation Intensive Pre-filter for String Pattern Matching in Network Intrusion Detection Systems,” *IEEE GLOBECOM 2006*, San Francisco, USA, November 2006.
[9] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep packet inspection using parallel Bloom filters,” *IEEE Micro*, vol. 24, no. 1, 2004, pp. 52–61.
[10] 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 (FCCM)*, Napa, CA, April 9–11, 2003, pp. 31–38.
[11] F. Yu , R. H. Katz , and T. V. Lakshman, “Gigabit rate packet pattern-matching using TCAM,” *IEEE International Conference on network protocols (ICNP’04)*, Oct. 5–8, 2004, pp. 174–183.
[12] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small Forwarding Tables for Fast Routing Lookups,” *ACM SIGCOMM’97*, Cannes, France, Sep. 1997, pp. 3–14.
[13] P. Gupta, S. Lin, and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” *IEEE INFOCOM’98*, San Francisco, CA, Apr. 1998, pp. 1240–1247.
[14] N. F. Huang and S. M. Zhao, “A Novel IP Routing Lookup Scheme and Hardware Architecture for Multi-Gigabit Switch Routers,” *IEEE Journal on Selected Areas in Communications (IEEE JSAC)*, vol. 17, no. 6, pp. 1093–1104, Jun. 1999.
[15] A. Donnelly and T. Deegan, “IP route lookups as string matching,” *IEEE International Conference on Local Computer Networks (LCN)*, 2000, pp. 589–595.
[16] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” *IEEE INFOCOM’04*, Mar. 2004, pp. 333–340.
[17] M. Norton, “Optimizing Pattern Matching for Intrusion Detection,” Sourcefire, Inc., Columbia, MD, Tech. Rep., Jul. 2004. Available: *http://www.sourcefire.com*
[18] T. Nishimura, S. Fukamachi, and T. Shinohara, “Speed-up of Aho-Corasick Pattern Matching Machines by Rearranging States,” IEEE SPIRE’01, Laguna de San Rafael, CHILE, 2001, pp. 175–185.
[19] J. van Lunteren, "High-Performance Pattern-Matching for Intrusion Detection," *IEEE INFOCOM'06 ,* Barcelona, Spain, Apr, 2006.
[20] V. C. Ravikumar and R. N. Mahapatra, “TCAM Architecture for IP Lookup Using Prefix Properties,” *IEEE Micro*, vol. 24, no. 2, 2004, pp. 60–69.
[21] H. Lim, J. Seo, and Y. Jung, “High Speed IP Address Lookup Architecture Using Hashing,” *IEEE Communications Letters*, vol. 7, no. 10, Oct. 2003, pp. 502–504.
[22] J. Crowcroft, “10 networking papers: recommended reading,” *ACM SIGCOMM Computer Communication Review*, Issue 2, Apr. 2006, pp. 31–32.
[23] [Online]. Available: *http://www.defcon.org*