Improved TCAM-based Pre-Filtering for Network Intrusion Detection Systems

Yeim-Kuan Chang, Ming-Li Tsai and Cheng-Chien Su
Department of Computer Science and Information Engineering
National Cheng Kung University
{ykchang, p7694157, p7894104}@mail.ncku.edu.tw

Abstract—**With the increasing growth of the Internet, the explosion of attacks and viruses significantly affects the network security. Network Intrusion Detection System (NIDS) is developed to identify these network attacks by a set of rules. However, searching for multiple patterns is a computationally expensive task in NIDS. Traditional software-based solutions can not meet the high bandwidth demanded in current high-speed networks. In the past, the pre-filtering designed for NIDS is an effective technique that can reduce the processing overhead significantly. A *FNP*-like TCAM searching engine (*FTSE*) [5][6] is an example that uses an 2-stage architecture to detect whether an incoming string contains patterns.**

In this paper, we propose two techniques to improve the performance of *FTSE* that utilizes ternary content addressable memory (TCAM) as pre-filter to achieve gigabit performance. The first technique performs the *w*-byte suffix pattern match instead of using *w*-byte prefix. The second technique finds the matching results from all groups rather than first group. We finally present the simulation result using Snort pattern set and DEFCON packet traces.

I. INTRODUCTION

With the growth of the Internet, large number of viruses and malicious probes spread every day. Many network hosts are vulnerable to attacks. Traditionally, networks have been protected using firewalls that monitor and filter network traffic. Firewalls usually examine the packet headers to determine whether the packets are allowed to go through or dropped. For example, if a packet attempts to connect a disallowed port, such as port number 80 (http), the connection is rejected. However, firewalls are not effective to protect network from worms and viruses. Today, the most commonly used defense strategy is to use end-host based solutions that rely on security tools, such as antivirus software. The main drawback of these approaches is the inability to protect thousands of hosts in less than an hour.

Network intrusion detection systems (NIDS) are utilized to detect malicious attacks and protect Internet system. The intrusion detection systems are growing in popularity because they can provide an efficient protection against the attacks. A NIDS differs from a firewall in that it needs to scan both the headers and the payloads of each incoming packet for thousands of suspicious patterns. By inspecting both packet headers and payloads to identify attack signatures, NIDS is able to discover malicious attacks or hackers that intend to intrude.

Because most of the known attacks can be represented with patterns or combinations of multiple sub-patterns, pattern matching has become a performance bottleneck in NIDS. Current NIDS pattern databases contain thousands of patterns, resulting in a difficult computational task. Traditionally, software-based NIDS may be overloaded when the packet arrival rate becomes high. To keep up with the high-speed networks, hardware-based NIDS implementation is needed.

Snort [9] is an open source light-weight NIDS which is designed to filter packets by pre-defined rules. Snort contains a set of rules with corresponding actions. Each Snort rule consists of the rule header and the rule options. The rule header contains the action, protocol, source and destination IP addresses, and the source and destination ports. The rule options contain alert messages and signatures which would be inspected to determine if the incoming packets match the rule. A sample Snort rule that detects *mountd* access is shown in Figure 1. When the new viruses or malicious attacks are discovered, corresponding rules are added to Snort. Because of its free availability and efficiency, Snort is quite commonly used and there are very large and current databases of signatures maintained on the Internet.

```
alert tcp any any -> 192.168.1.0/24 111
(content:"|00 01 86 a5|"; msg: "mountd access";)
```

Figure 1. Sample Snort rule

The rest of the paper is organized as follows: In section 2, we review and summarize the related works. In section 3, we propose two approaches to improve TCAM pre-filter techniques. The simulation results are presented in section 4. Finally, we present the conclusions in section 5.

II. RELATED WORKS

In the past few years, several interesting algorithms and techniques have been proposed for multiple-pattern matching. Current software-based NIDS cannot meet the bandwidth requirement of a multiple-gigabit network. Hence, several hardware-based solutions have been proposed to solve the problem. In this section, we divide the common pattern matching solutions for intrusion detection system into four categories: software-based, FGPA-based, bloom filter based, and TCAM based. The detail of each solution will be discussed as follows.

A. Software-based solutions

Knuth-Morris-Pratt (KMP) [4] and Boyer-Moore (BM) [2] are the most well-known single-pattern matching

1550-445X/08 $25.00 © 2008 IEEE
DOI 10.1109/AINA.2008.120

algorithms. Current Snort implementations use Boyer-Moore algorithm because BM has the best average-case performance. However, the performance of the BM algorithm depends on the characters in the text and pattern strings. The time complexity of the BM algorithm is sub-linear of *O(n/m)* in the best case, *O(n)* in the average case, and *O(nm)* in the worse-case . The Aho-Corasick(AC) [1] is designed for multiple-pattern matching. By pre-processing the patterns and build a finite state machine, AC can process the input text in linear time. However, a large transition table will slow down the performance in practice.

B. FPGA-based solutions

Many hardware-based algorithms have been proposed, where many solutions are based on Field Programmable Gate Arrays (FPGAs). Because of the emergence of new worms and viruses, traditional intrusion detection systems must be able to be re-programmed. FPGAs can be programmed for fast pattern matching due to their exploitation of reconfigurable hardware capability and their ability for parallelism. In recent years, there are several researches on FPGA pattern-matching for NIDS. Sourdis *et al.* proposed a FPGA-based approach by using pre-decoding technique [7]. The main idea of pre-decoded CAM is to test for equality of the input for the desired characters instead of keeping a window of input characters in the shift register. The results in [7] show the best throughput is about 11Gbps on Xilinx Virtex2 devices.

C. Parallel Bloom filters

Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element or string is a member of a set. False positives are possible, but false negatives are not. Dharmapurikar *et al.* [3] proposed a multiple-pattern matching solution using parallel bloom filters. The proposed scheme builds a bloom filter for each possible pattern length. The Bloom filter engine obtains the input by reading a data stream that arrives at the rate of one byte per clock cycle. However, the hardware cost becomes a problem because each different pattern length requires a separate bloom filter. The hardware cost problem becomes worse when dealing with very long virus definitions. Furthermore, this scheme needs to inspect every single byte of packet payload so that the maximum throughput is around 2.46Gbps [3].

D. TCAM solutions

Ternary Content Addressable Memory (TCAM) is a type of memory that consists of a set of entries. A TCAM allows fully parallel search of entries per TCAM lookup. Each TCAM entry can store three values for every bit: zero, one and "don't care". Don't care bits act as wildcards during a search. The top entry of the TCAM has the smallest index and the bottom entry has the largest. The width of each entry can be configured according to user requirements. For example, a 1M TCAM can be programmed as 64K entries with 16 bytes per entry, or 32K entries with 32bytes per entry. Given an input string, TCAM can compare the input with all the entries in parallel and report the searching result with just one TCAM lookup time.

Fang Yu *et al*. [8] proposed a TCAM-based pattern matching algorithm for handling both short patterns and long patterns. If a pattern is shorter than TCAM width, then we put it into TCAM and pad it with don't care bits. A pattern longer than TCAM width will split into several sub-patterns: the first TCAM width prefix pattern and remaining suffix patterns. There are three data structures to be stored in memory for matching long patterns: pattern table, partial hit list, and matching table. To match a pattern, a window of characters from input string is looked up in the TCAM. Then, the result is stored in a temporary table. The window is moved forward by a character and the lookup process is executed again. At every stage, the approximate partial match table entry is taken into account to verify if a complete pattern is matched.

III. PROPOSED ALGORITHMS

A. A brief introduction to FTSE

The basic concept of *FTSE* [5][6] is described as follows. FTSE first reads the first w bytes of the data stream as the input called the *sliding window*. The sliding window in next cycle is obtained by advancing 1 to w bytes of the data stream, depending on how the sliding window in the current cycle matches the patterns. If any i-byte suffix of sliding window does not match the i-byte prefix of pattern P for all $i = 1$ to w, we can advance the sliding window by skipping the current w bytes of the data stream and continue the search with the next w bytes. Specifically, if the i-byte suffix of the sliding window does match the i-byte prefix of pattern P for all $i = 1$ to $w - 1$, then we only skip $w - i$ bytes to get the sliding window for the next cycle and repeat the search process. If the sliding window matches the w-byte prefix of a pattern, we send the remaining $m - w$ bytes of the pattern of width m into an additional matching device called *exact matching module* for exact matching starting at the character next to the sliding window of the data stream. However, in this case, the next sliding window in the next cycle is obtained by advancing only one byte of the data stream. This is because if the exact matching fails (e.g., *false positive*), it is still possible that we can find the match starting at the second character of the sliding window of the data stream.

We now describe how FTSE works and its architecture. The TCAM is divided into w groups from G_0 to G_{w-1}, where w could be set to the width of TCAM. Group G_0 stores the w-byte prefixes of patterns, group G_1 stores the concatenation of one don't-care byte and (w–1)-byte prefixes of patterns, group G_2 stores the concatenation of two 'don't care' bytes and (w–2)-byte prefixes of patterns, and so on. If the pattern is shorter than w bytes (denoted as *short pattern*), we pad it with don't-care bytes. Figure 2 shows an example of *FTSE* TCAM layout for pattern *P='ABCDEFG'* with TCAM width $w = 4$.

The process of finding patterns in the payload of a packet is as follows: The first w bytes of the packet's payload is fetched as the initial sliding window and then looked up against the patterns stored in TCAM. If the sliding window matches an entry in group G_0, we need to perform the following two additional operations. First, if the pattern is shorter than TCAM width, then the full pattern is found. Second, if the pattern is longer than TCAM width, an exact match should be performed next by shifting the sliding window one byte. If the sliding window matches an entry in group G_i for $1 \le i \le w-1$, the sliding window will shift $w - i$ bytes to get the next sliding window for processing. On the other hand, if there is no

G0	ABCD
G1	*ABC
G2	**AB
G3	***A

Figure 2. TCAM layout for a pattern *P='ABCDEFG'* with TCAM width *w=4*.

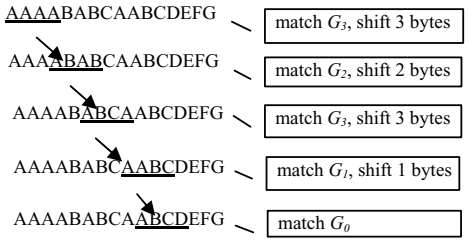

Figure 3. An FTSE example that needs five TCAM lookups.

match in TCAM, the sliding window will be shifted by *w* bytes. Figure 3 shows an example of matching process. For a pattern *P='ABCDEFG'*, TCAM with width of four is organized as in Figure 2. We first search the first 4 bytes of payload in TCAM and find a match in group G_3. Therefore, the sliding window is shifted by one byte. By repeating this search process, we find a match with five TCAM lookups and then report the pattern ID to exact matching module.

Figure 4 shows the hardware architecture of *FTSE* which consists of two parts: TCAM pre-filter module and exact matching module. The incoming data stream is first filtered through TCAM pre-filter module, which matches *w*-byte prefixes of patterns. If a match occurs in group G_0, the corresponding ID of the partial matching pattern is sent to the exact matching module for performing the exact matching between the potential pattern and input data stream. The controller determines the shift value of sliding window according to lookup results in TCAM. The full patterns are stored in 'Pattern Table'.

To verify the effectiveness of *FTSE*, we analyze the TCAM memory requirement and theoretical expected shift value which is defined to be the expected number of skipped characters per cycle. If we set TCAM width to *w*, then the *(w-i)*-byte prefix of each pattern is stored in group G_i for $0 \le i \le w-1$. Suppose we have a total of *N* patterns and the number of prefix-patterns in group G_i is N_i. The total number of TCAM entries is ΣN_i. So the total TCAM memory requirement is $w \times \Sigma N_i$ bytes.

To calculate the average shift value, we need to calculate the matching probability in each group of the TCAM. For a random *w*-byte input stream in TCAM, the matching probability of an entry in group G_i is

$$P(G_i) \cong \frac{N_i}{(2^8)^{w-i}},$$

where the number of don't-care bytes in G_i is *i* and each character is 8 bits The probability of mis-match in one TCAM lookup is

$$1-\sum P(G_i).$$

Therefore, the average shift value S_{avg} is

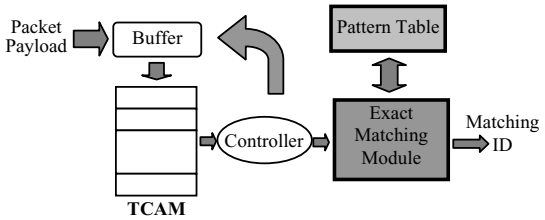

Figure 4. The hardware architecture of *FTSE*.

$$S_{avg} = \left(1-\sum P(G_i)\right)\times w + \sum \left(P(G_i)\times i\right),$$

Increasing the TCAM width greatly reduces the probability of false positive and increases the average shift value. The performance of *FTSE* depends on the average shift value which is the expected number of TCAM lookups needed to match the sliding window against the patterns in TCAM. However, it results in a huge amount of TCAM memory. Thus, choosing *w* is a tradeoff between cost and performance.

According to our observation, the probability of finding a match in group G_{w-1} is greater than that in other groups. As we mentioned earlier, the average shift value can be increased by reducing the probability of finding a match in TCAM. On the other hand, if we can increase the mis-match probability in one TCAM lookup that results shifting *w* bytes in the sliding window, the total number of TCAM lookups for packet payload should be reduced. In the following subsections, we provide two approaches for improving the efficiency of TCAM pre-filter.

B. Suffix matching

As stated, the don't-care bytes in TCAM entries increase the probability of finding a match in TCAM. For example, the probability of matching an entry '***A' in one TCAM lookup is $1/2^8$ which is large. The average shift value will decrease while the number of entries in group G_{w-1} increases. But, in this case, the number of TCAM lookups will also increase before we can find a match in group G_0 and thus we can send the partial matching result to the exact matching module. In summary, it is better to get a mis-match in TCAM as often as possible, so that the sliding window can skip *w* bytes and the exact matching module does not get involved. For this reason, our improved schemes try to avoid using the don't-care bytes in TCAM as much as possible.

Our first improved approach matches *w*-byte suffixes of patterns instead of prefix matching. We divide the TCAM entries into *w* groups from G_0 to G_{w-1}, similar to *FTSE*. If the patterns are shorter than *w*, they will be stored in TCAM in the same manner as *FTSE*. Otherwise, for a pattern of length *m* denoted by *P*[*1…m*], group G_i stores the *w*-byte suffix of sub-pattern *P*[*1…m – i*] for $i = 0$ to $w-1$. If the sub-pattern *P*[*1…m – i*] is shorter than *w*, we also pad it with don't-care bytes and store it in group G_i.

Sub-patterns that belong to the same group should be organized according to their lengths in descending order. It is because TCAM only reports the first matching result among multiple matches. In this way, patterns whose lengths are longer than or equal to *2w-1* occupy one TCAM entry in all groups without padding any don't-care byte. Figure 5 shows an example of our improved TCAM layout for a pattern *P='ABCDEFG'* and TCAM width

G0	DEFG
G1	CDEF
G2	BCDE
G3	ABCD

Figure 5. TCAM layout for a pattern *P='ABCDEFG'* with TCAM width *w=4*.

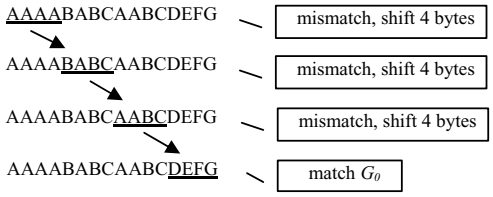

Figure 6. An example of the sliding window shift with four TCAM lookups

w=4. Since the don't-care byte does not exist in any TCAM entry, the probability of matching each TCAM group is the same. Therefore, the throughput can be improved due to less needed TCAM lookups. Figure 6 shows an example of matching process. Comparing to Figure 3, we find a match with only four TCAM lookups. The proposed suffix algorithm is shown in Figure 7.

Notice that the search process in TCAM is the same as *FTSE*. That is, when we have a match in group G_i for $i = 1$ to $w - 1$, we shift the sliding window by i bytes and continue the search with the new sliding window. When we get a match in group G_0, we trigger the exact match module by sending the partial matching result to it.

Since the don't-care state could not be eliminated entirely for some shorter patterns, we might have several types of entries in each group depending on how many don't-care bytes they have. The entries that contain inclusive relation in the same group could be eliminated without losing accuracy. For this reason, we can remove redundant entries from all groups except group G_0. For example, the group G_i ($i \neq 0$) contains three entries: *'ABCD'*, *'*BCD'*, and *'***D'* in Figure 8. Since the input stream matches the entry 'ABCD' or '*BCD', it also matches the entry '***D'. Therefore, we remove the entries 'ABCD' and '*BCD' from group G_i. Figure 8 also shows the result after removing the redundancy.

C. Multi-character processing

The *FTSE* or our improved scheme determines how many bytes to shift the sliding window in input data stream according to the matching result in TCAM. The number of characters skipped in sliding window varies from one to *w*. This kind of variation in skipping different number of characters results in an unstable throughput and needs an extra hardware support like buffer to control the sliding window. We extend the concept of subsection III-B to provide a high performance pre-filter approach which can shift multi-character per TCAM lookup.

Modern TCAM provides a blocking feature that divides a TCAM into several blocks and allows users to selectively search one or several blocks in parallel. With this feature, different groups can be put into different

Input: pattern $P = p_1 p_2 \ldots\ldots p_m$, and TCAM width w
Output: w entries
Suffix algorithm
{
  if ($m \geq w$)
  {
    for $i = 0$ to $w-1$ {
      if $(m\text{-}pos) >= w$
        store $p_{(m-w+1)-i} \ldots\ldots p_{m-i}$ into group G_i
      else //we must pad it with 'don't care' byte
        store *…* $p_1 \ldots\ldots p_{m-i}$ into group G_i
    }
  }
  else we handle it as *FTSE*
}

Figure 7. Proposed suffix algorithm.

Gi	ABCD
	*BCD
	***D

Gi	***D

Figure 8. An example of removing redundant entries.

G0	BCDE
G1	ABCD
G2	*ABC
G3	**AB

Figure 9. TCAM layout for a pattern *P='ABCDE'* and TCAM width *w=4*.

blocks of the same TCAM and the matching result of each group can be reported separately.

The basic idea of the second proposed pre-filter approach is to find the final partial matches from all the groups rather than only G_0, so that we can send the final partial match to the exact matching module for performing the exact matches. Since a series of *w*-byte sub-patterns are stored in all the groups, our approach determines a match from any one of these groups, instead of only the group G_0. The group ID along with the offset of the partial match position in the corresponding matched pattern will be sent to exact match module. Take Figure 5 as an example. If input data stream matches the entry 'CDEF' in group G_1, it will generate a match signal and send an offset value that match second four-bytes suffix of pattern *P* to the exact matching module. In this way, we can process multi-character per clock cycle.

However, this approach suffers from high false positive that generating by the certain of patterns. Those patterns, whose lengths are shorter than $2w - 1$, are padded with one or more don't-care bytes, especially the group G_{w-1}. The problem with this approach is that short patterns cause a TCAM matching frequently so false positive probability will increase greatly. For example, in Figure 9, the pattern *P='ABCDE'* was divided into four sub-patterns according to our suffix matching approach with TCAM width *w=4*. Those sub-patterns, 'BCDE', 'ABCD', '*ABC' and '**AB', were stored into groups from G_0 to G_3, respectively. The probability of false positive matches increases greatly in this example as the probability of matching sub-pattern '**AB' increases.

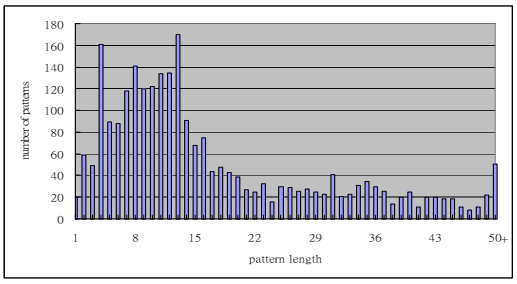

Figure 10. Length distribution of patterns in the Snort databases.

We can solve this problem by both increasing TCAM width and restricting the number of groups less than *w*. With a large *w* and a small number of entries in a group, we can greatly reduce the probability of finding a match in TCAM. Let *w'* be our new TCAM width and *g* be the reduced number of groups in TCAM. Since each pattern is divided into *g*×*w'*-byte sub-patterns in series, our approach requires *g*× *w'*×*N* bytes of TCAM memory approximately and can process *g* characters per TCAM lookup, where *N* is the total number of patterns. Although reducing the number of groups in TCAM also restricts the performance, the simulation results shows that our proposed scheme is better than *FTSE*.

IV. EXPERIMENTS

In this section, we present the experimental results of our two improved approaches. We utilized Snort pattern sets and DEFCON [10] packet traces to evaluate the effectiveness of the proposed algorithms and original *FTSE*.

We used the Snort [9] version 2.4 for our experiments. The total number of unique patterns in Snort is 2535. The pattern length distribution is shown in Figure 10. The average pattern length is 17 bytes and the maximum pattern length is 364 bytes.

We use five DEFCON8 packet traces for our TCAM pre-filter simulations. "RootFu!" is a contest run at DEFCON each year. "Capture the Capture the Flag" (CCTF) is a project by "The Shmoo Group" to sniff and log all the data on the "RootFu!" network. Our packet traces can be derived from the CCTF project held in DEFCON. Table I shows the matching statistics of these five DEFCON8 packet trace. The highest pattern-match rate in our experiments is in the packet trace 4 and the lowest one is in the packet trace 2.

The throughput in our TCAM pre-filter architecture will be increased as the average shift value becomes high. It is because the total number of TCAM lookups can be decreased. Increasing the TCAM width significantly reduces the amount of false positive and increase the average shift value. However, TCAM memory is one of the more expensive components in this architecture. Reducing the amount of required TCAM memory is a major concern. Therefore, we simulate our improved TCAM pre-filter approaches by using several TCAM widths.

Table II shows the TCAM memory size requirement with TCAM width 4, 8, 12, and 16. We observe that our suffix matching approach had less TCAM memory requirement when TCAM width is 12 or 16. It is because that we eliminate certain inclusive entries from group *1* to group *w-1*.

Table I. DEFCON8 packet traces.

Trace Name	Total payload size (bytes)	# of matched events	Percentage
Defcon8_trace1 (28153903)	148,878,041	15,110,119	10.1%
Defcon8_trace2 (28160701)	13,434,830	781,811	5.8%
Defcon8_trace3 (29124449)	86,117,534	6,306,690	7.3%
Defcon8_trace4 (29142147)	78,488,243	8,972,176	11.4%
Defcon8_trace5 (29190000)	159,480,998	11,834,997	7.4%

Table II. TCAM memory size (KB)

	FTSE	Suffix
4	12	14
8	80	89
12	218	195
16	432	332

Table III. TCAM width impact on Avg. shift value for DEFCON8 trace1.

	FTSE	Suffix
4	2.693	2.707
8	4.032	4.012
12	4.744	4.712
16	5.144	5.144

Table IV. TCAM width impact on Avg. shift value for DEFCON8 trace2

	FTSE	Suffix
4	3.075	3.092
8	4.847	4.858
12	6.011	6.014
16	6.796	6.796

Table V. TCAM width impact on Avg. shift value for DEFCON8 trace3

	FTSE	Suffix
4	2.921	2.955
8	4.529	4.539
12	5.470	5.473
16	6.061	6.062

Table VI. TCAM width impact on Avg. shift value for DEFCON8 trace4

	FTSE	Suffix
4	2.619	2.641
8	3.859	3.842
12	4.477	4.451
16	4.819	4.819

Table VII.TCAM width impact on Avg. shift value for DEFCON8 trace5

	FTSE	Suffix
4	2.965	2.985
8	4.650	4.664
12	5.665	5.693
16	6.333	6.322

For every packet trace, we measure the average shift values, as shown in Table III to Table VII. With the TCAM width increasing by four sequentially, the average shift value increases slowly. The results shows that our suffix approach had better average shift value than FTSE in shorter TCAM width, 4 especially. It is because the number of short patterns becomes low. By comparing with different packet traces, we can get higher throughput than FTSE when pattern-match rate of packet trace is low.

Since the DEFCON traces tend to match patterns in nature, the pattern-match rate is higher than normal packets. We believe our suffix approach can be improved if the packet traces are normal ones instead of DEFCON traces.

For our second approach with multi-character processing, we perform the experiments by using different sets of TCAM width *w'* and the reduced number of groups

Table VIII. *Parallel* (*w'*, *g*) with *w'*=2*g*

	Parallel 6-3	Parallel 8-4	Parallel 10-5	Parallel 12-6
TCAM memory (KB)	31	59	89	121
Avg. Shift (byte)	3	4	5	6

Table IX. *Parallel* (*w'*, *g*) with fixed TCAM width 8 and varied g

	Parallel 8-3	Parallel 8-4	Parallel 8-5	Parallel 8-6
TCAM memory (KB)	43	59	75	90
Avg. Shift (byte)	3	4	5	6

g, denoted as *parallel(w', g)*. Table VIII and Table IX show the simulation results with different combinations of *w*' and *g*. When *g* increases, the TCAM memory and average shift value will increase. Besides, the false positive was serious due to the short patterns. In our experiments, *w*' is double of the value *g* would be a proper choice. The results show that our second approach is very efficient especially for malicious packets.

V. CONCLUSION

In this paper, we proposed two effective TCAM-based pattern matching approaches for high-speed networks. We used Snort pattern sets and DEFCON packet traces to evaluate the performance. Our TCAM-based approach provides two techniques to improve the performance of *FTSE*. The first technique matches the *w*-byte suffixes of patterns instead of *w*-byte prefixes. The second technique finds the final partial matching results from all the groups instead of only G_0. The second proposed scheme can process multi characters per TCAM lookup. The results showed that our two techniques can are better than the original *FTSE*.

REFERENCES

[1] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," Communications of the ACM, vol. 18, no. 6, pp.333-343, June 1975.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no 10, pp.762-772, Oct. 1977.

[3] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull and J. W. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters", IEEE Micro, vol. 24, no. 1, pp. 52-61, Jan. 2004.

[4] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," SIAM J. Comput., vol. 6, no. 2, pp. 323–350, June 1977.

[5] R.T. Liu, C.N. Kao, H.S. Wu, M.C. Shih and N.F. Huang, "FTSE: The FNP-Like TCAM Searching Engine," in IEEE Symposium on Computers and Communications, pp 863-868, June 2005.

[6] R.T. Liu, N.F. Huang, C.H. Chen, C.N. Kao, "A Fast String Matching Algorithm for Network Processor-Based Intrusion Detection System", ACM Transactions on Embedded Computing Systems, Vol. 3, No. 3, Aug. 2004, pp. 614-633.

[7] I. Sourdis and D. Pnevmatikatos. "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching." In IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM), Napa, CA, Apr. 2004.

[8] F. Yu, R. H. Katz, T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in Proceeding of 12th IEEE International Conference on Network Protocols (ICNP'04), Berlin, Germany, Oct. 2004, pp. 174-183.

[9] Snort - the de Facto Standard for Intrusion Detection/Prevention, http://www.snort.org

[10] DEFCON http://cctf.shmoo.com/data/cctf-defcon8/ and http://www.shmoo.com/