

# DFA-based Regular Expression Matching on Compressed Traffic

Yan Sun and Min Sik Kim

School of Electrical Engineering and Computer Science

Washington State University

Pullman, Washington 99164-2752, U.S.A.

Email: {ysun,msk}@eecs.wsu.edu

**Abstract**—Many network security applications in today’s networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet. For example, traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, viruses, attack signatures, etc. Regular expressions are often used to represent such patterns. They are implemented using finite automata, which take the payload of a packet as an input string. However, existing approaches, both non-deterministic finite automata (NFA) and deterministic finite automata (DFA), do not deal with compressed traffic, which becomes more and more popular in HTTP applications. In this paper, we propose an efficient algorithm for regular expression matching to implement deep packet inspection on compressed traffic. Based on the observations of DFA, we design a scheme to skip most of the matching process in the compressed parts of traffic. To the best of our knowledge, this is the first effort to design an efficient regular expression matching on compressed traffic. We evaluate our algorithm using rule sets provided by Snort, a popular open-source intrusion detection system. The evaluation results show that our approach can reduce the number of state access in the DFA significantly.

## I. INTRODUCTION

Many network security applications in today’s networks are based on deep packet inspection, checking not only the header portion but also the payload portion of a packet. Traffic monitoring, layer-7 filtering, and network intrusion detection all require an accurate analysis of packet content in search for predefined patterns to identify specific classes of applications, viruses, attack signatures, etc. Those patterns were traditionally a number of strings representing signatures to be compared against packet contents using exact matching algorithms. However, exact matching is not expressive enough to detect malicious patterns with the evolution of the network threats and evasion techniques. Thus, more expressive regular expressions are used to describe a wide variety of payload signatures [1]. For example, in the Linux Application Protocol Classifier [2], all protocol identifiers are expressed as regular expressions. Similarly, Snort [3], which is an open-source network intrusion detection system, has evolved from no regular expressions in its rule set in April 2003 to 3,737 unique Perl Compatible Regular Expressions (PCRE) as of November 2009. Bro [4], another open-source intrusion detection system, also uses regular expressions as its signature language. These

regular expressions are also used in commercial firewalls and other networking equipments including Cisco’s IOS [5].

The most popular method to implement regular expression matching is to use finite automata [6]–[8]. In this method, a finite automaton is built based on the given regular expressions, and is run with packet payload as input. The finite automaton is either deterministic or non-deterministic, depending on underlying technologies and available resources. A non-deterministic finite automaton (NFA) requires as many state transitions per character in the payload as the number of states in the worst case, and thus has a time complexity for each character of  $O(m)$ , where  $m$  is the number of states in the NFA. However, it is very efficient in terms of space usage compared to a deterministic counterpart. These properties make NFAs more suitable for ASIC (application-specific integrated circuit) or FPGA (field-programmable gate array) implementations, which can provide wide bandwidth but small amount of on-chip memory. On the other hand, a deterministic finite automaton (DFA) requires only one state transition per character, while it needs a much larger amount of memory for the same regular expression. Therefore, DFAs are more suitable for general-purpose processors and network processors.

LZ77 [9] compression algorithm is commonly used in today’s Internet traffic, and the basic idea of the LZ77 compression algorithm is that when a series of continuous characters (a substring) has already appeared in the near past, we can use a pair of numbers to represent this repeated substring. The pair of numbers is also called a length-distance pair which represents the distance in bytes of the two substrings and the length in bytes of the repeated substring. We can also call such a pair of numbers a pointer and call the compressed substring pointer area. For example, the plan text: abcdefgabcde will be compressed into: abcdefg(7, 5). When performing regular expression matching on such compressed traffic, the naive approach is to decompress the traffic first and then perform regular expression matching on the plan text as usual, but this process is costly because the deep packet inspection itself consumes a lot of CPU time and memory resources. So we propose an efficient algorithm to perform regular expression matching on compressed traffic based on the properties of compression algorithm and DFA.

The remainder of the paper is organized as follows. In

Section II, related work in regular expression matching is presented. Section III provides a discussion on using finite automata for regular expression matching, and Section IV describes our algorithm and its implementation. Then, the proposed algorithm is evaluated in Section V. Finally, we conclude in Section VI.

## II. RELATED WORK

Nowadays, regular expressions are the language of choice in NIDS (network intrusion detection systems) from commercial vendors such as 3Com's TippingPoint X505 [10] and Cisco IPS [5], as well as open source NIDS such as Bro [4] and Snort [3]. Those regular expressions are typically implemented using finite automata, either NFA or DFA. Because a DFA requires at most one state transition per input character, it is faster than a NFA in many cases, and thus has been a preferred way to implement regular expression matching. For the thousands of complex regular expressions as found in Snort's rule sets, the DFA implementation consumes prohibitive amount of memory. We divided the approaches to regular expression matching into the following three categories:

*a) ASIC-based:* Several commercial network equipment vendors, including 3Com [10] and Cisco [5] have supplied their own NIDS and a number of smaller players have introduced pattern matching ASICs which go inside these NIDS. ReCPU is a fast ASIC-based regular expression matching approach [6]. In fact, many have argued that deep packet inspection should happen in ASICs. Developing ASICs for NIDS, however, has several disadvantages; it requires a large investment and a long development cycle, and it is hard to upgrade.

*b) FPGA-based:* There is a body of literature advocating FPGA-based pattern matching [11]–[14]. It can provide not only fast matching cycle but also parallel matching operations. NFAs are well-suited for FPGA-based matching because of its wide bandwidth requirement and low memory consumption. Pre-decoded CAMs [16] and Bitwise optimized CAM [17] are FPGA-based architectures that use character pre-decoding coupled with CAM-based patterns to accelerate the matching speed. This type of approaches, however, is not flexible enough for general-purpose regular expression matching. Besides, it is still expensive and power-consuming.

*c) Software-based:* The software-based approaches are also called general-purpose approaches, and they are based on general-purpose processors or network processors [7], [8], [18]–[21]. DFAs are more popular in software-based approaches because they only need one state transition per input character, which causes at most one memory access for each character input. However, As we mentioned earlier, the practical use of DFAs is limited because of their excessive memory usage. In order to mitigate this issue, many methods have been proposed [18]–[21]. They develop several memory compression techniques for DFAs, focusing on reducing the number of transitions between states, and in some cases, 99% transitions can be eliminated.

Actually, all the previous works focus on how to use NFA/DFA more efficiently but without considering compressed traffic. [22] focuses on this issue but only considers simple length-fixed pattern matching, and we will further explore efficient algorithm to perform deep packet inspection on compressed traffic based on regular expressions, which are more complex and commonly used in today's Intrusion Detection Systems.

## III. REGULAR EXPRESSION MATCHING AND FINITE AUTOMATA PROBLEMS IN COMPRESSED TRAFFIC

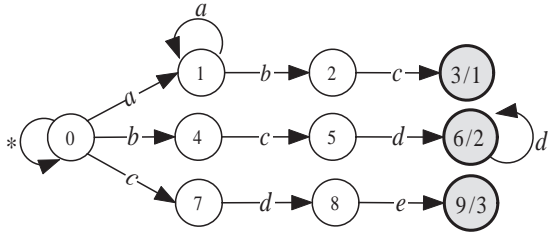
A network intrusion detection system (NIDS) classifies packets using a predefined rule set to determine whether packets are malicious or not by searching packet payloads for any signature in the rule set. Because of the increasing amount of network traffic and threats, intrusion detection systems become very resource-intensive. For instance, open-source NIDSs such as Bro [4] and Snort [3] expend all the resources, both CPU time and memory, and halt immediately when they are deployed under high-speed network environment [23]. Therefore, achieving high-throughput in regular expression matching and reducing memory access frequency are crucial for overall intrusion detection performance. The regular expression matching becomes the bottleneck in the network applications and exhausts the CPU and memory resources, and the decompression procedure further degrade the overall performance, so how to reduce the frequency of state access (memory access) is important and our goal is to skip as many input characters as possible to improve performance.

As we introduced, the finite automaton is either non-deterministic or deterministic; a non-deterministic finite automaton (NFA) requires as many state transitions per character in the payload, which means there can be multiple active states at the same time, and this makes NFA not suitable for checking the compressed traffic, because it is very difficult to record previous state sequence. While the DFA only has one active state all the time, which makes its state sequence easily to be recorded and reused, and that's why we choose DFA to represent regular expressions instead of NFA for compressed traffic.

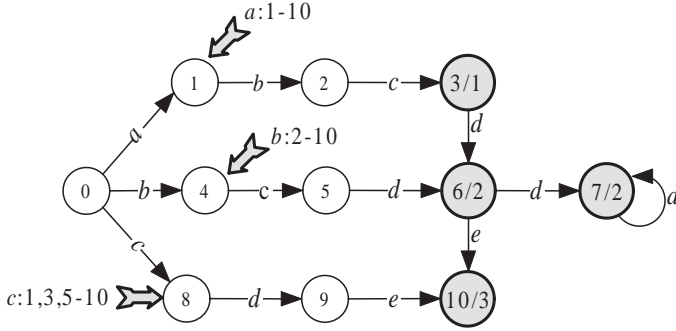
## IV. OUR APPROACH

### A. Observation on DFA-Based Regular Expression matching

There have been many efforts to reduce memory consumption in DFA implementations. In [8], Kumar et al. developed the Delayed Input DFA (D<sup>2</sup>FA), which reduces space requirements by reducing the transitions based on the observation that many states have similar sets of outgoing transitions. In D<sup>2</sup>FA, such transitions are grouped and represented by a single default one. In this way, their algorithm achieves a reduction of memory consumption by more than 95%. However, the drawback of this approach is that it may need to process multiple states for a single input character, which increases overall memory bandwidth. Later, Ficara et al. proposed Delta Finite Automata in [18]. It is based on the same observation as D<sup>2</sup>FA, but only requires a transition per input character.



(a)



(b)

Fig. 1. NFA in (a) and DFA in (b) based on regular expressions (1)  $a + bc$  (2)  $bcd+$  (3)  $cde$

We revisit this from another viewpoint. Those redundant transitions usually occupy more than 95% of total transitions, which means, given an input character, it is very likely that the next state is the same state, regardless of the current state. Based on these observations, we propose an algorithm to perform regular expression matching on compressed traffic.

To clarify these properties of DFA, we analyze the same example brought by Becchi et al. in [24]. In Fig. 1, we show the NFA and DFA accepting regular expressions  $a + bc$ ,  $bcd+$ , and  $cde$  on the alphabet  $\Sigma = \{a, b, c, d, e\}$ , constructed in the standard way [25]. Transitions leading to state 0 are omitted. The big arrows represent transitions leading to the current state.

We show two examples of skipping the entire pointer area and skipping part of the pointer area in Fig. 2.

We define terminologies used in Fig. 2 as follow:

- 1) C-Trf: Compressed traffic including pointers
- 2) D-Trf: Decompressed traffic without pointers
- 3) State: State number in the DFA
- 4) Status: The status of state in the DFA, unmatched (u) or match (m)

In the DFA in Fig. 1, we can see that with the same input character, the possible next states are very limited, and they are shown in Table I.

From Table I we can see that with the same input character, it is likely to reach the same state, even from different current states.

### Example 1:

C-Trf:  $e b c d a e c b \{6, 6\} c$   
D-Trf:  $e b c d a e c b c d a e c b c$   
State: 0 4 5 6 1 0 8 4 - - - - - 4 5  
Status: u u u m u u u u - m - - - - u

### Example 2:

C-Trf:  $e b c d e a b c \{6, 6\} c$   
D-Trf:  $e b c d e a b c c d e a b c c$   
State: 0 4 5 6 10 1 2 3 8 9 10 - - 4 5  
Status: u u u m m u u u - - m - - - u

Fig. 2. Skip the entire pointer area in Example 1 and skip part of the pointer area in Example 2

TABLE I  
STATES COMPARISON BETWEEN NFA AND DFA

Input character	Next state	ratio between these states
$a$	1	1
$b$	2, 4	1:10
$c$	3, 5, 8	1:1:9
$d$	6, 7, 9	2:2:1
$e$	10	1

Most of the time, the active state is the original state, which is usually state 0, or states reachable from the original state with one input character. In this example, state 0, 1, 4 and 8 are such states. In our experiment, for more than 70% of time (1361 out of 1835) the active state was one of these states. That is because traffic seldom matches attack signatures in practice. There are 257 such states including state 0, because there are 256 symbols represented with 8 bits, and they create 256 states as the next states from state 0. If the destination state is in this set of states, the same input character will lead all states to the same state. In other words, the next state is deterministic and it does not matter what the current state is. We call the part of DFA including these states the Deterministic Area. Note that 95% of transitions will lead to this area as we discussed above. In the example shown in Fig. 2, there are a total of 56 transitions and 46 of them lead to states in the Deterministic Area, which means the probability of going into the Deterministic Area is about 82% on average in this example. With the high probability for the current state to be in the Deterministic Area, the probability of entering the Deterministic Area in the next step is even higher in practice. We call the rest of DFA the Non-deterministic Area. Note that with the same input character, it is possible to reach the same state from the different current states, even in the Non-deterministic Area, e.g.,  $3 \rightarrow 6$  and  $5 \rightarrow 6$ . With more attack traffic, there will be more matches, and there will be more activities in the Non-deterministic Area, because it is likely to go deeper in the DFA. The Deterministic Area and Non-deterministic Area in a DFA are defined as follows:

- Deterministic Area: The starting state (state 0) and the

*C-Trf*: e c d e a b {5, 3} c  
*D-Trf*: e c d e a b c d e c  
*State* : 0 8 9 10 1 2 3 6 10 8  
*Status* : u u u m u u m m m u

Fig. 3. The worst case in Fig. 1 caused by path pairs in our algorithm

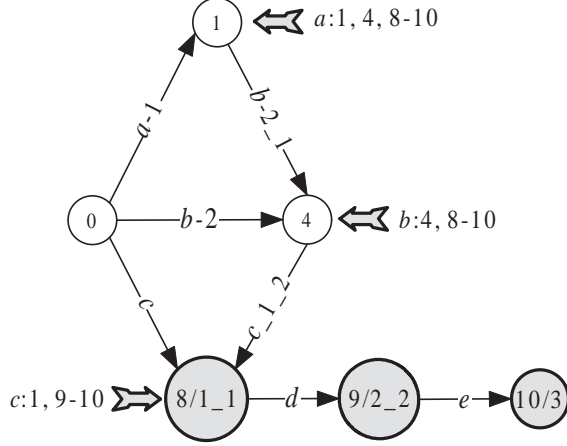


Fig. 4. The compressed DFA for the example in Fig. 1(b)

states reachable from the starting state directly.

- Non-deterministic Area: States that are not in the Deterministic Area.

If we need to check the pointer area for a long substring and cannot skip these characters, there must be at least two paths in the DFA that accept the same input substring. We call these two paths a *path pair*. With today's DFA compression technique, however, the number of path pairs is decreasing significantly. In other words, each path is more likely to be unique, because the redundant parts have been combined or reduced. This fact helps us to skip more characters in compressed areas.

For example, in the DFA in Fig. 1, the worst case is caused by such path pairs:  $2 \Rightarrow 3 \Rightarrow 6 \Rightarrow 10$  and  $0 \Rightarrow 8 \Rightarrow 9 \Rightarrow 10$ , with the same input string "bcd," which is shown in Fig. 3. In this example, we need to check the entire pointer area again.

Based on the fact that the path pairs degrade our algorithm, we propose a method to reduce the path pairs by combining the path pairs. Based on the same example shown in Fig. 1(b), we combine the two sets of path pairs, and the results are shown in Fig. 4. We use tokens to represent different paths, and a token is created following a dash symbol, in our example, a token is maintained following an underline symbol in the transitions. An existing token will be removed if it does not meet a maintaining symbol during a transition. There could be multiple tokens at the same time. And the state 8 or state 9 can be a match state if one of the current tokens matches the token in these states (token 1 and token 2 in state 8 and state 9 respectively). So after the compression, there is no path pairs in our example and the worst case is eliminated, then all

*C-Trf*: e c d e a b {5, 3} c  
*D-Trf*: e c d e a b c d e c  
*State* : 0 8 9 10 1 4 8 - - 8  
*Token* : - - - - 1 1,2 1,2 2 - -  
*Status* : u u u m u u m m m u

Fig. 5. The worst case in Fig. 3 after path pairs compression

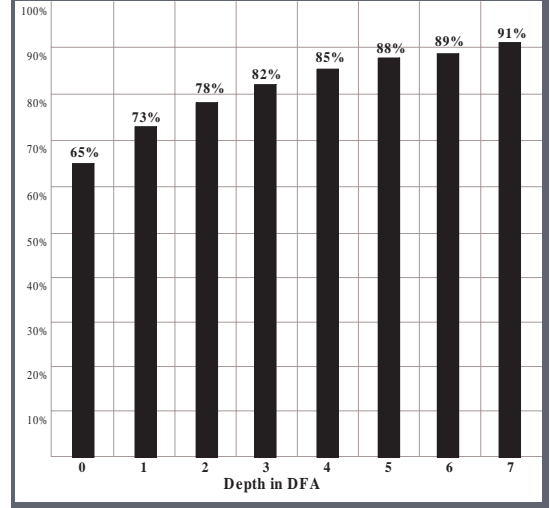


Fig. 6. The probability to enter the same state as the previous string

characters excluding the first one in the pointer area can be skipped. The worst case shown in Fig. 3 is shown in Fig. 5 after path pair compression.

## V. EVALUATION

In our experiment, the deeper we check the pointer area the more chances we go into the same state as previous substring, and the relationship is shown in Fig. 6. The 0 value in x-axis represents the probability we can skip the whole pointer area. We can see that after checking seven characters in the pointer area, we have more than 90% of probability to skip the rest of the pointer area.

We selected five groups of traffic with different compression ratios from 10% to 90%, and build DFAs using three different regular expression sets  $R_1$ ,  $R_2$  and  $R_3$  with 100 regular expressions, 200 regular expressions and 400 regular expressions respectively collected from Snort rule set. We simulate the numbers of DFA state accesses in the original approach and the numbers of DFA state accesses in proposed approach, and the simulation results are shown in Table II. And the percentages of DFA state accesses saved by our approach are shown in Fig. 7.

From the evaluation results we can see that our approach can achieve desired goal and the percentage of DFA state accesses we can save is mainly based on the compression ratio of the traffic but has little relationship with the number of regular expressions used. Furthermore, there is no false positive or false negative in our approach.

TABLE II

PERCENTAGES OF DFA STATE ACCESSES WE CAN SAVE WITH DIFFERENT COMPRESSION RATIOS USING 100 REGULAR EXPRESSIONS

Compression ratio	10%	30%	50%	70%	90%
Number of original DFA state accesses	1364	1473	1753	1835	1583
Number of our DFA state accesses in $R_1$	1240	1083	1015	795	469
Number of our DFA state accesses in $R_2$	1241	1092	1029	804	505
Number of our DFA state accesses in $R_3$	1244	1093	1011	815	503

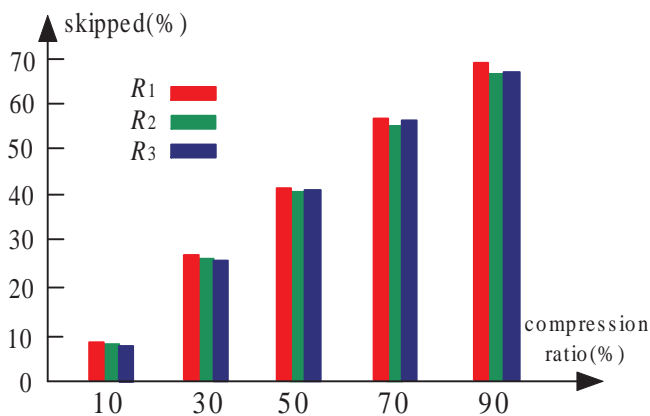


Fig. 7. Percentages of DFA state accesses saved by our approach

## VI. CONCLUSION

In this paper, we proposed an efficient regular expression matching in deep packet inspection for compressed traffic. We first analyzed the properties of NFA and DFA, and chose DFA due to its one active state property. Based on the observation that with the same input character there is a high probability that the next state will be the same state regardless of the current state, we built an efficient DFA generator by combining some states and transitions to further improve this property. Our algorithm records previous information of previous active state sequence and matching sequence, then some compressed parts of the traffic can be skipped if the active state also repeat. The simulation results show that our approach can efficiently skip most of the compressed parts in the traffic and reduce the frequency of DFA state accesses based on the traffic's compression ratio. Because DFA is not suitable for a large number of complex regular expressions [26], our approach can be easily extended to be used in separated multiple small DFAs, and that makes our approach well scalable.

## REFERENCES

- [1] R. Sommer, V. Paxson, and T. Mnchen, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Oct. 2003.
- [2] J. Levandoski, E. Sommer, and M. Strait, "Application layer packet classifier for Linux," <http://l7-filter.sourceforge.net>.
- [3] *Snort User Manual 2.8.6*, The Snort Project, Apr. 2010, available as [http://www.snort.org/assets/140/snort\\_manual\\_2\\_8\\_6.pdf](http://www.snort.org/assets/140/snort_manual_2_8_6.pdf).
- [4] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.

- [5] "Cisco IOS IPS signature deployment guide," <http://www.cisco.com/>.
- [6] M. Paolieri, I. Bonesana, and M. D. Santambrogio, "ReCPU: A parallel and pipelined architecture for regular expression matching," in *Proceedings of 15th Annual IFIP International Conference on Very Large Scale Integration*, 2007, pp. 19–24.
- [7] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2007, pp. 1–12.
- [8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006, pp. 339–350.
- [9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, pp. 337–343, May 1977.
- [10] "TippingPoint X505," [http://www.tippingpoint.com/products\\_ips.html](http://www.tippingpoint.com/products_ips.html).
- [11] I. Bonesana, M. Paolieri, and M. Santambrogio, "An adaptable FPGA-based system for regular expression matching," in *Proceedings of the conference on Design, Automation and Test in Europe*, 2008, pp. 1262–1267.
- [12] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *International Conference on Field Programmable Logic and Applications*, Sep. 2008.
- [13] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003, pp. 31–38.
- [14] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in *Conference on field-programmable logic and applications*, 2003, pp. 956–959.
- [15] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001, pp. 227–238.
- [16] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Processing of IEEE Symposium On Field-Programming Custom Computing Machines*, 2004, pp. 258–267.
- [17] S. Yusuf and W. Luk, "Bitwise optimised CAM for network intrusion detection systems," in *Processing of IEEE Symposium On Field-Programming Custom Computing Machines*, 2005, pp. 444–449.
- [18] D. Ficara, S. Giordano, G. Proccisi, F. Vitucci, G. Antichi, and A. D. Pietro, "An improved DFA for fast regular expression matching," *Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.
- [19] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, Dec. 2007.
- [20] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, Dec. 2006.
- [21] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proceedings of the 26th IEEE International Conference on Computer Communications*, May 2007, pp. 29–40.
- [22] A. Bremler-Barr and Y. Koral, "Accelerating multi-patterns matching on compressed HTTP traffic," in *Proceedings of the IEEE INFOCOM 2009*, 2009, pp. 397–405.
- [23] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational experiences with high-volume network intrusion detection," in *Proceedings of the 11th ACM Conference on Computer and Communication Security*, Oct. 2004.
- [24] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 11th ACM Conference on Computer and Communication Security*, Oct. 2004.
- [25] J. E. Hopcroft and J. D. Ullman, "Introduction to automata theory, languages, and computation," *Addison Wesley*, 1979.
- [26] Y. Sun, H. Liu, V. Valgenti, and M. S. Kim, "Hybrid regular expression matching for deep packet inspection on multi-core architecture," in *Proceedings of the 19th International Conference on Computer Communications and Networks, ICCCN'10*, Aug. 2010, pp. 1–7.