2009 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools

Methodology for Fast Pattern Matching by Deterministic Finite Automaton with Perfect Hashing

Jan Kastil
CESNET z.s.p.o
Zizkova 4, Prague, 160 00
Czech Republic
kastil@liberouter.org

Jan Korenek
Faculty of Information Technology
Brno University of Technology
korenek@fit.vutbr.cz

Ondrej Lengal
CESNET z.s.p.o
Zizkova 4, Prague, 160 00
Czech Republic
xlenga00@liberouter.org

Abstract—**As the speed of current computer networks increases, it is necessary to protect networks by security systems such as firewalls and Intrusion Detection Systems operating at multigigabit speeds. Pattern matching is the time-critical operation of current IDS on multigigabit networks. Regular expressions are often used to describe malicious network patterns. This paper deals with fast regular expression matching using the Deterministic Finite Automaton (DFA) with perfect hash function. We introduce decomposition of the problem on two parts: transformation of the input alphabet and usage of a fast DFA, and usage of perfect hashing to reduce space/speed tradeoff for DFA transition table.**

I. Introduction

In recent years, Internet has become a very popular method to connect computers all over the world. While the availability of continuous communication has created many new opportunities, it has also brought new possibilities for malicious users. The importance of network security is therefore growing; one of the ways of malicious activity detection on a network is by using Intrusion Detection Systems (IDS).

Most modern IDS rely on a set of rules that are applied to each input packet in order to define suspicious activities. The simplest rules are described by packet header field content and pattern of data in the packet payload. Detecting such patterns is the core operation of an IDS.

Modern IDS cannot process each packet independently, because an attacker can split the string described by the pattern between multiple packets. To identify such attacks, IDS must scan each network flow as one stream. For stream reconstruction, some information is needed to be stored for every flow. As the number of flows can be high and the memory size is limited, the stored information needs to take minimum size. If a DFA is used for pattern matching, the flow does not need to be reconstructed. It is sufficient to store the last state of the DFA at packet boundary and continue pattern matching from the stored state when the next packet of the flow arrives.

Many papers deal with the problems described above, but none of them was able to present a method sufficiently fast for wire-speed processing on multigigabit networks. The main problem of suggested methods are memory requirements. We propose to use a perfect hash function to implement the transition table of the DFA and reduce memory requirements to the minimum without sacrificing any advantage of DFA-based methods. Our method can be used with optimization methods [1],[2] that reduces the number of transitions in a DFA.

The paper is divided into following sections: Section II briefly mentions related work for pattern matching, while in Section III we introduce the concept of alphabet transformation to accelerate pattern matching. Section IV proposes perfect hash function to implement DFA transition table. In Section V we describe synthesis of the regular expression into hardware matching units. Section VI describes experimental results obtained by evaluation of our methodology, and finally, Section VII concludes our work and suggests next possible ways of our research.

II. Related Work

The problem of fast pattern matching is addressed by many researchers. Therefore, many methods which are suitable for fast pattern matching has been introduced, but according to our knowledge, there is no algorithm that optimally addresses all requirements of a modern IDS. First IDS used only string-based patterns, but Sommer and Paxton noted in [3] that patterns based on regular expressions can be more effective than pattern based only on strings. String matching algorithms are fast but their extension to regular expressions is not always possible. TCAM [4] or KMP [5] algorithms could be considered examples of such methods. Methods for string matching that can be extended into matching of patterns described by regular expressions are often based on Finite Automata.

The drawbacks of methods based on FA is that a FA can accept only one character per transition and parallelization of FA itself still remains to be solved. There are two major approaches to pattern matching using FA. The first group of methods use Nondeterministic Finite Automaton. Clark et al [6] used NFA and obtained the throughput of 100 Gb/s.

Using this approach, NFA needs to be synthesized into an FPGA from each set of patters from a hardware description language, such as VHDL or Verilog. Therefore, fast change of the matching pattern is not possible, which limits its deployment into HW acceleration of an IDS, because if a new type of attack occurs, adequate rule has to be added immediately into the IDS. The Witty Worm [7], for example, was able to infect the majority of vulnerable hosts in about 45 minutes. Another approach to NFA implementation uses backtracking to find a correct transition path through the automaton. This approach cannot be used in an IDS, because its time complexity is worse than linear.

Another approach to FA-based methods is the use of Deterministic Finite Automaton. DFA can be implemented to run with linear time complexity. Due to the fact that DFA can be in only one active state, its transition table can be stored in the memory instead of being implemented in the logic. Implementing the transition table in memory allows to change the sets of patterns without the need for reconfiguration and reduces the time of change. The speed of on-chip memory becomes the limiting factor in these implementations. For successful deployment of DFA-based methods into an IDS, it is crucial to minimize the memory requirements for the DFA. This problem is addressed by many researchers: [8], [9], [10], [2]. Unfortunately, many of these method were primarily developed for string matching and their properties on a set of patterns described by regular expressions has not been fully examined.

An interesting combination of both approaches is a method suggested by Gonzalo Navarro and Mathieu Raffinot in [9], where Glushkov's algorithm is used for construction of a NFA. In Glushkov's automaton, all incoming transitions to one state are labelled by the same symbol. Therefore, it is possible to represent such automaton as a pair of bit tables. The first table maps each symbol onto a bit vector of the length of the state set size. Each bit of the vector represents one state of the automaton and the whole vector represents all states which are accessible by the specified character. The second table has a row for each possible bit vector from the first table. Each such line contains a set of states which are accessible from the given set of states in the form of a bit vector. When a transition occurs, a logical `AND` operation is performed on result vectors from both tables. The resulting vector represents a set of states in the NFA or one state in the DFA. The drawback of this method is the size of the state because each state of the DFA has to be represented by a bit vector of the size equal to the size of the state set of the corresponding NFA. In the IDS systems, it is necessary to scan the whole network flow, therefore the active state of the automaton needs to be stored with each network flow.

Nathan Tuck et al in [8] implement the transition table as a tree-like structure. Each state is represented by a bitmap for all possible input characters and a pointer to the next state. Then a transition from such state while reading the current symbol exists only when the bit reserved for the current character is set to one in the bitmap. If a transition exists, then the sum of all ones before the current character is added to the pointer and the result is the address of the next state. It is obvious that this method is suitable only for string matching and it is not possible to be used for regular expression matching without changes [11], because it does not allow to implement a cycle in the automaton, and therefore Kleene closure and positive closure cannot occur in the pattern.

A string matching method introduced in [12] comes from the observation that most of network traffic is not malicious activity and that the automaton is able to recognize such traffic in several first steps. Therefore, the authors suggest to accept more than one character in a transition from the start state in order to increase the throughput. Although the autors present the method only for string matching problem, it can be extended also for regular expressions.

III. Fast Regular Expression Matching

Regular expression matching engines have to examine every character of the input stream. If modern backbone networks are to be examined, more than one billion characters per second need to be processed. Because currently available general purpose processors are limited to GHz frequency and a small number of instructions processed in one clock cycle, they cannot achieve requested throughput. Therefore FPGA implementations, which enable high pipelined parallelism, are widely used. Unfortunately, the clock frequency of an FPGA differs with different designs and cannot exceed 250 MHz. Therefore, a pattern matching unit working at 250 MHz has to examine at least 4 characters per clock cycle to achieve the throughput of 10 Gb/s.

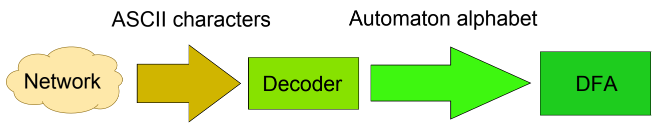

Fig. 1. Two basic steps of our methodology

Fig. 1 shows two basics steps of our methodology. The first step is alphabet transformation which encodes n-tuples of ASCII characters into one symbol of the automaton input alphabet. The results of the transformation enters the DFA for pattern matching. Both units are pipelined and work simultaneously.

More formally, let Σ_i be the alphabet of the input stream and Σ_o be the input alphabet of the automaton. Let $S = \{x \mid x \in (\Sigma_1)^n\}$, where n is the number of characters accepted in one clock cycle. The transformation T from the alphabet Σ_i to Σ_o is defined as $T \subset S \times \Sigma_o$.

The size of the memory necessary to represent the DFA transition table depends on the number of states and the size of the input alphabet. This means that it is important to reduce the size of the alphabet. This can be achieve by introducing character classes [13]. Character class is the set of all characters which is used to label the transition. A transition in the DFA is made if and only if the accepted character belongs into the character class. An example of the character classes is shown in Fig. 2.

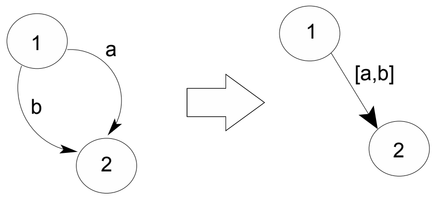

Fig. 2. Character classes

A DFA can accept only one symbol in a clock cycle and, therefore, the decoder has to assign exactly one output value to each possible input string. The alphabet transformation which meets such demand is called *deterministic transformation*. An alphabet which can be obtained by deterministic transformation is called *deterministic alphabet*. Every nondeterministic alphabet can be transformed to its deterministic version at the cost of increasing the number of symbols in the alphabet. The detailed description of the construction of alphabet transformation is beyond the scope of this paper. We refer to [14] for the algorithm to construct the deterministic alphabet as a proof of concept. Additional details concerning implementation of the decoder are given in Section V.

IV. Perfect Hash Look-up Methodology

Despite the use of character classes in alphabet transformation, multicharacter automaton has extremely sparse and large transition table. Moreover, the automaton often represents more than one pattern and the transition table is even larger. As the automaton has to process the input traffic at wire speed, the transition has to be performed within a few clock cycles, therefore a look-up operation for the next state has to be performed in constant time. Fig. 3 shows the basic idea of the FA implementation. The position in the transition table block computes pointer into the transition table, where the actual transition is stored. The result of this look-up enters the validate block, which decides whether the found transition belongs into the automaton.

Many methods for size reduction of the transition table have been studied, e.g. [2] or [1], so that the transition table could be store in smaller and faster memory. Despite the fact that these methods yield good results, the transition table still remains large and sparse.

Look-up in a large and sparse transition table has to be performed in constant time in order to achieve wire speed processing of modern network traffic. Moreover, the memory overhead of the look-up algorithm has to be reasonably small. Hash tables are good candidates for implementation of transition tables because of constant time complexity of look-up operation. However, a drawback is the possibility of collision, which leads to more than one memory access per transition. We propose to use *perfect hash function* to eliminate collisions in the transition table. A single memory access for look-up is needed in the worst case. The perfect hash function look-up will work only for transitions that belong to the automaton. When other combination appears at the input of the perfect hash function, the result is undefined. In fact, the result of the perfect hash function will be the position of some existing transition. Therefore, the membership of the transition needs to be tested before changing the state of the automaton.

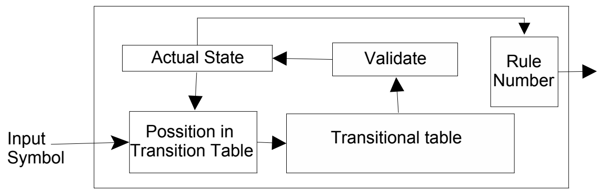

Fig. 3. Representing DFA

A. Use of perfect hashing to find the new active state

A DFA is always in exactly one active state. At the beginning of the stream, the active state of the automaton is its start state. Representations of the active state and the input symbol are put together. The created bit string represents at most one transition in the DFA and is hashed by perfect hash function found in the preprocessing step. The result is used as an index into the transition table. Because perfect hash function does not have any collisions, only two situations can occur: (*i*) the transition exists in the automaton, then the data stored at the computed position represents the new active state, (*ii*) the transition does not exist in the automaton and therefore the pattern could not be found in the stream. It is obvious that only one memory access into the transition table is needed to determine the new active state.

B. Test of transition validity

As it has been mentioned in the previous section, perfect hash function (PHF) does not have collisions between two transitions in a FA. However, if the input combination does not exist in the transition table, PHF will return an invalid result. Therefore, the next state is obtained in a single memory access but the validity of the transition needs to be checked. If the transition is invalid, default transition has to be applied. It can be a transition to initial, final or any internal node of FA.

Deciding transition validity requires to store some additional information and the size of this additional information determines memory efficiency of this approach. The straightforward approach would be to store the key of the hash function with the new state of the automaton. Then the process of validation is implemented as comparison of the stored value and the key of the hash function. The disadvantage of this approach is large size of the key. Early experiments has shown that the size of the key can be up to 32 bits, while the state can be stored only in a 10-bit word. Therefore, it is required to introduce some compression into storage of key values. Example of such compression by another hash function can be found in [15].

However, with the use of perfect hash function, the problem becomes more specific. A transition is valid if and only if it exists in the automaton and the transition table always returns a correct next state for a valid transition. Therefore, we propose to store only the next state in the transition table and to solve the problem of transition validity independently as a problem of set membership, which is very well studied ([16], [17]).

V. Automatic Regular Expression Synthesis

Patterns are described by regular expressions, however, regular expressions are not suitable for pattern matching unit configuration. The pattern matching unit requires a pattern in data structures which allow fast look-up. This section briefly describes the preprocessing phase of the pattern matching that transforms a group of regular expressions into a single DFA, which accepts multiple characters per transition.

Modern IDS contain hundreds or thousands of patterns which have to be tested. It is not possible to implement so much hardware units, and therefore some of the patterns need to be merged together into one matching unit. If all patterns would be merged into one unit, the resulting pattern would be extremely complicated and memory requirements for such pattern would be very high. Dividing the whole set of patterns into several groups [18] seems to be a feasible option. Methods that divide rules into several groups are beyond the scope of this paper and are addressed by many researchers.

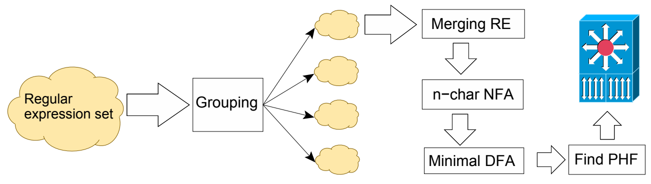

Fig. 4. Description of synthesis

Fig. 4 shows a basic diagram of regular expressions synthesis. The process starts with a set of regular expressions. Before the synthesis, this set is needed to be diveded into groups which will fit into matching units. Each such group is than processed by the same algorithm and downloaded into a separate hardware unit.

The first step is merging all rules in one group into one NFA without ϵ transitions. This is done by Thomson's construction described in [19]. The result of this algorithm is a simple transformation into ϵ-free form. The second step is extension of the automaton to accept more than one character per transition, and concurrently the alphabet transformation is also computed. The resulting NFA-epsilon is then determinized and minimized. The third step is finding the perfect hash function that will map transitions into memory. The last step prepares the automaton for hardware. It consist of two substeps:

- Preparing data structures for alphabet transformation
- Preparing the transition table

A. *Data structures for alphabet transformation*

Alphabet transformation can be seen as a classification problem. A hardware unit performing the transformation simply classifies input n-tuples into one character class. The problem of classification is very well studied in many research works, such as [20], [21], or [22]. Each of these algorithms requires different representation of the alphabet transformation. The main purpose of this step is to create suitable representation for hardware classification unit. The tree bitmap structure was chosen as a suitable representation, because it can be easily constructed and the classification works in linear time with the length of the transforming sequence and always returns correct results. Implementation of the tree bitmap can be pipelined and therefore an output symbol can be generated every clock cycle.

B. *Transition table*

The transition table is stored in a hash table using perfect hash. There are many algorithms for perfect hashing ([23], [24], [25], [15], [26], [27]); each one of them requires some time for finding perfect hash function, but when the PHF is found, its result can be computed in constant time. The biggest difference between these algorithms are memory requirements of the created PFH. According to the abovementioned studies, the algorithms in [25] and [26] have the smallest memory requirements. Unfortunately, the algorithm in [26] requires exponential preprocessing time in the worst case, while the algorithm in [25] completes preprocessing step always in linear time. Therefore, [25] is used for perfect hashing in the transition table. The perfect hash function can be found by this algorithm in a matter of seconds. When the perfect hash function is found, every transition has its own place in the transition table, which is filled with keys for the hash function and new active states of the automaton. This structure is then uploaded into hardware together with the perfect hash function.

The algorithm [25] is based on random hypergraphs. In the first step, three random hash functions are selected. These hash functions and set of keys (S) form random hypergraph. Each hash function maps every key from S into interval m $< \ 1, M \ >$, where $M \geq |S|$. More specifically, interval m is divided into subintervals of the same size and every hash function maps a key into its own subinterval. Every key is hashed by all three hash functions and their results represent one edge of the hypergraph. The hypergraph can be considered random, because hash functions are randomly generated. From random graph theory, it is possible to compute constant probability that a random hypergraph is going to be acyclic. The actual probability depends on the ration between size of S and M. If the ratio is above 1.23, then the probability of random graph being an acyclic hypergraph approaches 1. If a graph is acyclic, it is possible to select one vertex from each edge in such way that this vertex will not be selected in any other edge. This means that the number of this vertex is unique for the key and can serve as a value of the PHF. When the PHF is evaluated, it only identifies the unique vertex from the edge. To do so, it is required to store two bit information with every vertex. This information is computed during creation of the PHF from the hypergraph.

Test of the hypergraph acyclicity always requires linear time and its time complexity does not depend on the inputs of the algorithm. The same holds for the transformation of the hypergraph into the PHF. Therefore, all parts of the algorithm have exactly linear time complexity. The only tricky part is the choice of the hash functions which generates acyclic hypergraph. The acyclic hypergraph is found with certain probability, which is near 1, but there is still a small possibility of failure. If failure occurs, new hash functions are generated and therefore a new hypergraph needs to be tested for acyclicity. This can be repeated as long as needed or until the specified numbers of iteration is reached, but the chance of more than five failures in line is virtually impossible. Therefore, the worst case of the PHF generation is about $O(kn+n)$ where n is the number of keys and k is the maximal number of the iteration, but in average case it is only $O(n+n)$. As we see, the algorithm always performs in linear time.

VI. Experimental Results

We evaluated our methodology on regular expressions used in Snort [28] rule set for viruses. The set of virus rules was divided into several groups of the same size by the algorithm in [18]. All experiments were done with one of these groups.

The first experiment we conducted was on alphabet transformations. Table I and Fig. 6 show the way the size of a nondeterministic alphabet depends on the number of characters accepted in a single transition of the automaton for different regular expressions. The number of regular expression is the sequence number of the regular expression in the computed group. It is clear that the size of the alphabet grows linearly with the number of characters accepted per transition contrary to the growth of the transitional table caused by all possible combinations of characters at the input, which would be exponential. The results prove that alphabet transformation is suitable for real world patterns. The slope of the graph depends on the type of regular expressions.

Table II and Fig. 5 show memory requirements of an automaton created from selected regular expressions. The automaton accepts two characters per transition. The number of the regular expression is the sequence number of the regular expression in the computed group. The second column represents the memory needed to store transition table in a memory array addressable by pair *(Symbol, State)*. Density of the table then represents the portion of the table filled with useful data. The fourth column contains the number of transitions in the automaton. The fifth column is the size of the transition table without the overhead given by perfect hash function and information for answering membership queries for transitions. This column is the lower bound of the size of memory required for the transition table. The sixth column represents the size of memory required to store transition table with additional information for the decision about validity of the transition and perfect hash function itself.

The experiment results show that our methodology is not well suitable for small and simple regular expressions, because a simple regular expression is represented by a small transition table. A small transition table can be implemented more easily by other methods. Therefore, we merged all studied regular expressions into one and studied the same characteristics for the result as in the previous experiment. We extended this automaton to accept more characters than two in one step to show how memory requirements of the automaton depend on the number of accepted characters. The results are summarized in Table III and Fig. 7. The graph shows that our methodology scales well with the complexity of the regular expression and also with the number of accepted characters.

VII. Conclusions and Future Work

The main contribution of this paper is introduction of a new algorithm to simulate DFA with extremely large alphabets. The proposed algorithm uses perfect hash function for constant time look-up in a transition table with minimal memory overhead. The paper deals with the issue of efficient implementation of a DFA, which is the core part of many modern Intrusion Detection Systems or anti-virus systems. The second contribution of the paper is the overview of methodologis for extremely fast pattern matching. The proposed methodology is suitable for frequent and fast changes of patterns. Moreover, the proposed solution uses fixed hardware resources and only increases memory requirements. Therefore, it is quite possible

rule	1-char	2-char	3-char	4-char	5-char
1	5	10	14	17	20
2	5	10	13	15	17
3	35	171	269	370	486
4	13	20	24	28	32
merged	39	194	304	413	537

TABLE I
The size of the nondeterministic alphabet depends on the number of accepted characters

RE Number	Theoretical size	Density of the table	Transitions	Memory size without overhead	Memory size with overhead
1	2240	0.0317	71	355	1278
2	576	0,0903	52	208	832
3	313344	0.0027	841	6724	23548
4	2550	0.0227	58	290	1044

TABLE II
Descriptions of the used patterns

Number of characters	Transitions	Theoretical size [kb]	Memory size without overhead [kb]	Memory size with overhead [kb]
1	556	57	4	13
2	1937	540	15	54
3	7749	2590	61	232
4	26014	13332	208	832
5	136202	68675	1089	4767

TABLE III
Descriptions of the whole group

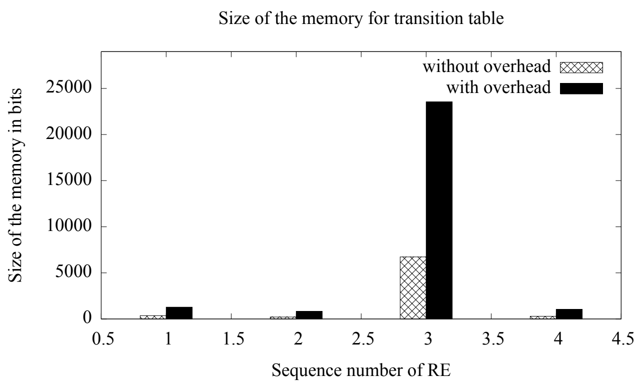

Fig. 5. Relation between memory size and type of regular expressions

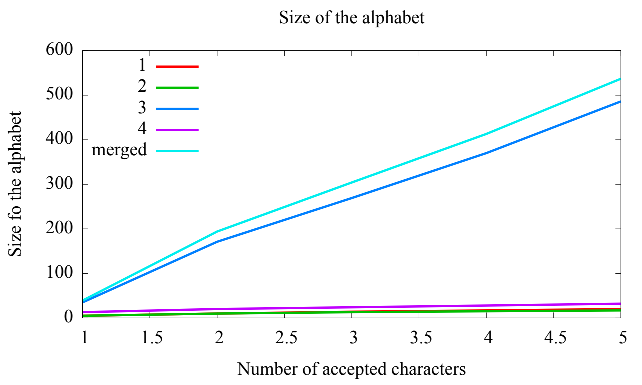

Fig. 6. Relation between the size of the alphabet and number of characters accepted per transition

to increase both the throughput and the number of patterns simply by using larger memory.

The immediate future work will concentrate on possible improvements of our methodology by more efficient alphabet transformation and introduction of additional compression into the transition table. The last but the most promising way of research seems to be implementing the ability to skip characters to the automaton. This would increase the average throughput but could decrease the worst case throughput.

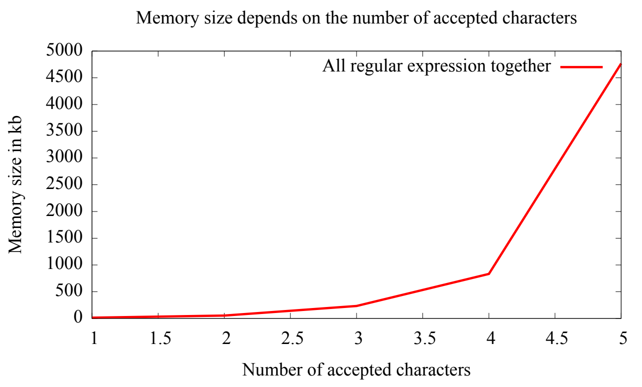

Fig. 7. Relation between memory size and number of characters accepted per transition

Acknowledgements

This research was supported by the Research Plan No. MSM, 0021630528 — Security-Oriented Research in Information Technology and Research Plan No. MSM, 6383917201.

References

[1] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," in *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2007, pp. 155–164.

[2] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2006, pp. 339–350.

[3] R. Sommer and V. Paxson, "Enhancing Byte-level Network Intrusion Detection Signatures with Context," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2003, pp. 262–271.

[4] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *ICNP '04: Proceedings of the 12th IEEE International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 174–183.

[5] D. E. Knuth, J. James H. Morris, and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977. [Online]. Available: http://link.aip.org/link/?SMJ/6/323/1

[6] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 249–257.

[7] C. Shannon and D. Moore, "The Spread of the Witty Worm ," *IEEE SECURITY and PRIVACY*, vol. 2, no. 4, pp. 46–50, 2004.

[8] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection," in *In IEEE Infocom, Hong Kong*, 2004, pp. 333–340.

[9] G. Navarro and M. Raffinot, "New Techniques for Regular Expression Searching," *Algorithmica*, vol. 41, no. 2, pp. 89–116, Nov. 2004.

[10] L. Tan, B. Brotherton, and T. Sherwood, "Bit-split String-matching Engines for Intrusion Detection and Prevention," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 1, pp. 3–34, 2006.

[11] R. Dixon, O. Eğecioğlu, and T. Sherwood, "Automata-theoretic analysis of bit-split languages for packet scanning," in *CIAA '08: Proceedings of the 13th international conference on Implementation and Applications of Automata*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 141–150.

[12] Y.-D. Lin, K.-K. Tseng, T.-H. Lee, Y.-N. Lin, C.-C. Hung, and Y.-C. Lai, "A Platform-based SoC Design and Implementation of Scalable Automaton Matching for Deep Packet Inspection," *J. Syst. Archit.*, vol. 53, no. 12, pp. 937–950, 2007.

[13] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation," in *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2007, pp. 145–154.

[14] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 191–202.

[15] S. Lefebvre and H. Hoppe, "Perfect Spatial Hashing," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*. New York, NY, USA: ACM, 2006, pp. 579–588.

[16] L. I. T. Pardo, "Set Representation and Set Intersection," Ph.D. dissertation, Stanford, CA, USA, 1978.

[17] P. Briggs and L. Torczon, "An Efficient Representation for Sparse Sets," *ACM Lett. Program. Lang. Syst.*, vol. 2, no. 1-4, pp. 59–69, 1993.

[18] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection," in *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM, 2006, pp. 93–102.

[19] K. Thompson, "Programming Techniques: Regular Expression Search Algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[20] H. Song, J. Turner, and J. Lockwood, "Shape Shifting Tries for Faster IP Route Lookup," in *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 358–367.

[21] X.-G. Wang, "Multi-Dimensional Packet Classification Using Tuple Space Based on Bit-Parallelism," *Intelligent Information Hiding and Multimedia Signal Processing, International Conference on*, vol. 0, pp. 197–200, 2006.

[22] W. N. Eatherton, P. Professors, S. L. Missouri, W. Eatherton, A. Professors, J. S. Turner, J. S. Turner, G. Varghese, and G. Varghese, "Hardware-based Internet Protocol Prefix Lookups."

[23] Z. J. Czech, G. Havas, and B. S. Majewski, "An Optimal Algorithm for Generating Minimal Perfect Hash Functions," *Information Processing Letters*, vol. 43, pp. 257–264, 1992.

[24] F. C. Botelho, Y. Kohayakawa, and N. Ziviani, "A Practical Minimal Perfect Hashing Method," in *In Proc. of the 4th International Workshop on Efficient and Experimental Algorithms (WEA 05)*. Springer, 2005, pp. 488–500.

[25] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and Space-efficient Minimal Perfect Hash Functions," in *In Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*. Springer LNCS, 2007, pp. 139–150.

[26] E. A. Fox, Q. F. Chen, and L. S. Heath, "A Faster Algorithm for Constructing Minimal Perfect Hash Functions," in *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*. New York, NY, USA: ACM, 1992, pp. 266–273.

[27] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect Hashing for Network Applications," in *in IEEE Symposium on Information Theory*. IEEE Press, 2006, pp. 2774–2778.

[28] "Snort homepage." [Online]. Available: www.snort.org