Hardware Accelerated Pattern Matching Based on Deterministic Finite Automata with Perfect Hashing

Jan Kastil, Jan Korenek
Faculty of Information Technology
Brno University of Technology
Bozetechova 2, Brno, 612 66, Czech Republic
Email: (ikastil, korenek)@fit.vutbr.cz

Keywords: Inrusion Detection, Perfect hashing, hardware acceleration, Deterministic Finite Automata

***Abstract*— With the increased amount of data transferred by computer networks, the amount of the malicious traffic also increases and therefore it is necessary to protect networks by security systems such as firewalls and Intrusion Detection Systems (IDS) operating at multigigabit speeds. Pattern matching is the time critical operation of current IDS. This paper deals with the analysis of regular expressions used by modern IDS to describe malicious traffic. According to our analysis, more than 64 percent of regular expressions create Deterministic Finite Automaton (DFA) with less than 20 percent of saturation of the transition table which allows efficient implementation of pattern matching into FPGA platform. We propose architecture for fast pattern matching using perfect hashing suitable for implementation into FPGA platform. The memory requirements of presented architecture is closed to the theoretical minimum for sparse transition tables.**

I. Introduction

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. In real networks, packets are often delivered out of the order. This situation is described and solved in [1].

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 [2],[3] that reduce 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 describe the implementation of the finite automata by the perfect hashing and the basic principle of selected perfect hashing algorithm. Section IV describes experimental results obtained by analyzing signatures from modern intrusion detection systems and finally, Section V concludes our work and suggest 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 have 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 demonstrated that patterns based on regular expressions can be more effective than pattern based only on strings [4]. String matching algorithms are fast but their extension to regular expressions is not always possible. TCAM [5] or KMP [6] algorithms could be seen as 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 an FA can accept only one symbol per transition and parallelization of FA itself still remains to be solved. It is possible to use alphabet transformation such that several input characters correspond to the one symbol of the automaton. These transformations can increase throughput of the solution but the core of the pattern matching unit still works sequentially. Even implementations of the nondeterministic FA can not accept more than one symbol per one step. Implementations of NFA allow only to minimize size of the automaton and therefore obtain higher frequency of the implementation.

There are two major approaches to pattern matching using FA. The first group of methods uses Nondeterministic Finite Automaton. Clark et al used NFA and obtained the throughput of 100 Gb/s [7]. 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 [8], for

978-1-4244-6613-9/10/$26.00 ©2010 IEEE

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: [9], [10], [11], [3]. 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.

III. Architecture

A. Perfect Hash Algorithms

There are many algorithms for perfect hashing ([12], [13], [14], [15], [16], [17]); 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 above mentioned studies, the algorithms in [14] and [16] have the smallest memory requirements. Unfortunately, the algorithm in [16] requires exponential preprocessing time in the worst case, while the algorithm in [14] completes preprocessing step always in linear time. Therefore, [14] is used for perfect hashing in the transition table. The perfect hash function can be found by this algorithm in a matter of seconds.

The algorithm [14] is based on random hypergraphs. In the first step, three random hash functions are selected. These hash functions and the set of keys (N) form random hypergraph. Each hash function maps every key from N into interval $u < 1, U >$, where $U \geq |N|$. More specifically, interval u 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 ratio between size of N and U. If the ratio is above 1.23 then the probability of random graph being an acyclic hypergraph approaches 1. If a hypergraph 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 works in linear time.

B. Architecture of Finite Automaton

The biggest problem of the DFA based implementations lies in the communication with the memory. Every accepted character requires communication with the memory. Moreover, the memory transition do not have any form of locality. Therefore it is not possible to use cache to speed-up the communication. DFA representation also consumes much more memory than its nondeterministic counter part. Many researchers focused on the minimization of the communication between memory and the DFA logic. While in [18] authors introduce small and fast memory that is only partially updated from external memory with whole transition table, [3] introduces Delayed DFA which is able to reduce size of the transition table for big automata at the cost of increasing number of transition needed to detect signature.

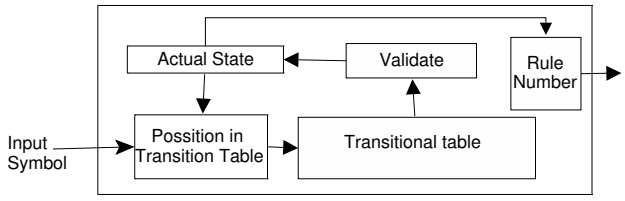

Fig. 1. Representing DFA

Fig. 1 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. Position in transition table is computed as a perfect hash function (PHF). Perfect hash returns unique address into the transition memory for every existing transition. Therefore it is possible to store only next states in the transition table

and achieve nearly 100 % utilization of the memory. Perfect hash function is computed from the input symbol and actual state of the automaton. Therefore it is possible to compute PHF from prohibited combination representing nonexistent transition. In such case, PHF will return undefined result. This result will be corrected by validate block. Validate block confirms if the key of PHF represents existing transition in the automaton. From the algorithmic point of view validate block solves membership problem if the transition belongs into the transition set. It is possible to parallelize computation of the perfect hash and validation process. It is required to validate transition in constant time. According to [19] the information-theoretic minimum number of bits required to store the set in the structure supporting membership queries in constant time can be computed by

$$B = \lg \binom{M}{N} \tag{1}$$

where N is the number of elements or transitions and M represents the universe or all possible transitions. By using Stirling's approximation, this formula can be replaced by

$$B = N \lg \frac{M}{N} \tag{2}$$

Authors of [19] also state that Perfect Hash Table is ideal representation for very sparse sets. With these arguments in mind, we propose to store the key of PHF in transition table together with next state in order to achieve high utilization of the memory and to allow storing of the whole transition table in the on-chip memory which allows random access to be performed in one clock. However, the size of the memory representation for this structure is bigger than the theoretical minimum. The real size can be computed by the formula

$$B = 1.23 * N * (2 + \lceil \lg |S| \rceil + \lceil \lg |\Sigma| \rceil) \tag{3}$$

where N is the number of transitions in the automaton and constant 1.23 is the minimal memory overhead for the perfect hash function [14]. Σ is the alphabet of the selected automata and $\lg |\Sigma|$ si the number of bits required to store one symbol, while S is the set of all states of the automata and $\lg |S|$ is the number of bits required to store one state of the automata. The constant of 2 bits is the memory required for PHF. In the implementation, next state is stored in the same memory line as the validation information, therefore the computation of the PHF, validation of the key and next state lookup can be done with one memory access during one clock step.

IV. Experimental Results

We evaluated our approach with the ruleset of Bro [20] and Snort [21]. Our idea assumes that transition table of the signatures is very sparse. Therefore the first experiment we conducted was to confirm this assumption. Fig.2 shows saturation histogram of the transition table of the DFAs for every signature in the Bro ruleset. The number of DFA is shown in logaritmic rate. The figure proves that most of the signatures have very sparse transition table and therefore are good candidates for our method. Fig.3 demonstrates saturation histogram of transition table of Delayed DFA presented in the [22]. It can be seen that transition table is less sparse than in previous figure. It shows that the Delayed DFA is suitable for much bigger automata than those in Bro ruleset.

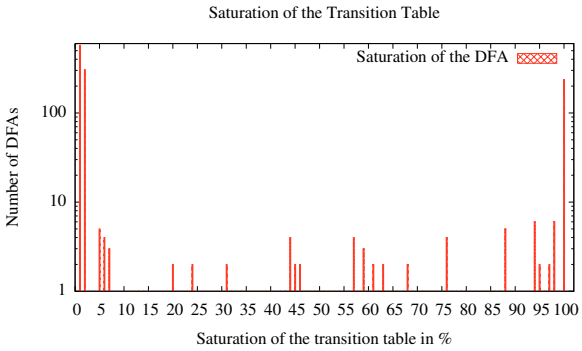

Fig. 2. Saturation of Transition Table of DFA

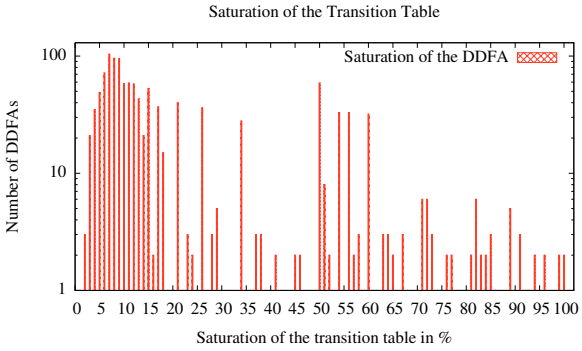

Fig. 3. Saturation of Transition Table of Delayed DFA

According to the presented histograms, more than 64% of the rules lesser saturation of transition table than 20%. In the successive experiments, we considered only the signatures with transition table saturation less than 20 percent. Remaining signatures should be implemented by another technique.

In the second experiment we focused on the memory consumptions of the validation block in the DFAs for different modules of modern intrusion detection systems. DFA was built for every rule in the module and DFA with saturation of transition table higher than 20 percent were removed. Information-theoretic size of the validation structure was computed according to the formula 1 and real memory required for proposed structure was computed according to the formula 3. Computation of combination number in formula 1 was problematic for some DFAs. In these problematic cases, approximation according to formula 2 was used. The last step consisted of the summarization of the results of all rules in module. Results of this experiment for every module with more than 30 signatures are presented in table I. The only exception is Snort module web-active, which does not contain any signature with saturation of transition table less than 20 percent. The Selected column informs how many percent

Module	Selected	States	Transitions	Theoretical bound	Real memory
Bro ex.webrules	94%	2898	8055	62 kb	96 kb
Bro Snort.default	68%	7232	19116	141 kb	233 kb
Snort back-door	64%	3150	15415	90 kb	217 kb
Snort spy-ware	18%	2637	12721	70 kb	168 kb
Snort web.client	40%	199	898	6 kb	10 kb
Snort web.misc	30%	212	1652	7 kb	20 kb

TABLE I
MEMORY REQUIREMENTS OF THE DFAS

regular expression of the module meet the saturation criteria. Other columns contain number of states and transitions in all automata in the module together with the sum of memory requirements of all dfas.

V. CONCLUSIONS AND FUTURE RESEARCH

The analysis of DFA transition tables for signatures used in intrusion detection systems is the main contribution of this paper. We used ruleset of programs Snort and Bro. The analysis shows that saturation of the transition tables changes from very sparse tables to high dense tables but 64% of all regular expression have DFA with saturation of the transition table less than 20%. According to presented experiments, multiple different algorithms have to be used to achieve optimal memory requirements. Therefore we focused on regular expressions with sparse transition table and designed a new architecture which is based on the perfect hashing. The proposed architecture was able to represent sparse tables in memory with capacity close to the theoretical minimal and can be easily mapped into FPGA.

In future work, we will continue in analysis of regular expressions. We want to analyse an influence of merging regular expressions to transition table size and density. We will also focus on multi-char automata in order to achieve higher throughput. We have the intention to identify effective implementation of multi-char automata with using alphabet transformation.

Acknowledgements

This research was supported by the Research Plan No. MSM, 0021630528 —Security-Oriented Research in Information Technology, Research Plan No.MSM, 6383917201 and the grant BUT FIT-S-10-1.

Authors wishes to thank Michela Becchi for the regular expression parser used in this work.

REFERENCES

[1] T. Johnson, S. Muthukrishnan, and I. Rozenbaum, "Monitoring regular expressions on out-of-order streams," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, April 2007, pp. 1315–1319.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[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] 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.

[17] 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.

[18] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, 2008.

[19] A. Brodnik and J. I. Munro, "Membership in constant time and almost-minimum space," *SIAM J. Comput.*, vol. 28, no. 5, pp. 1627–1640, 1999.

[20] V. Paxson, "Bro: a system for detecting network intruders in real time," in *In Computer Networks*, 1999.

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

[22] 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.