AN FPGA BASED ARCHITECTURE FOR COMPLEX RULE MATCHING WITH STATEFUL INSPECTION OF MULTIPLE TCP CONNECTIONS

Claudio Greco, Enrico Nobile, Salvatore Pontarelli, Simone Teofili

CNIT/University of Rome ”Tor Vergata”,
Via del Politecnico 1, 00191, Rome, ITALY
email: claudiogre@gmail.com, enrico_n@hotmail.com,
simone.teofili@uniroma2.it, salvatore.pontarelli@uniroma2.it

ABSTRACT

In this paper a novel architecture for string matching is presented. It is oriented to an FPGA implementation and, differently from other similar works, it is particularly suitable for rules matching in multiple streams. The paper presents our developed architecture able to efficiently manage different streams, discusses how to optimize the design to limit the number of FPGA logic resources and shows the obtained results.

1. INTRODUCTION

The continuous increasing of network speed and its diffusion in all the aspects of our life is constantly accompanied by an increasing request of network security. One of the most used solutions for protecting networks is based on the use of Network Intrusion Detection Systems (NIDS) such as Snort [1].

An IDS should perform several tasks to completely analyze the traffic crossing the network, reassembling TCP flows, collecting several statistics, classifying packets according to header or content matching, comparing the payload with content-matching rules. Two tasks can be considered of fundamental importance:

1. stream preprocessing: The TCP/UDP packets coming from the same connection are reassembled to provide to the the rule matching module an ordered data stream, allowing the analysis of the streams as an entire data flow, not packet by packet.

2. rule matching on a reordered stream: the stream is inspected in order to check if any of the IDS rules is matched. The identification of a malicious flow requires not only the matching of a particular string, but the subsequent identification of one of more suspicious contents (*e.g.* strings, bytes, regular expressions), placed in specific positions inside the data flow, that can be located in different packets.

IDS are usually software based and are used in little networks, far from the backbones links. In fact, due to the large amount of data that must be analyzed by a NIDS, the software implementation of a NIDS cannot sustain a traffic rate comparable with the rates available on a backbone. Therefore a lot of effort has been spent in the last years to implement NIDS in hardware, increasing their processing capabilities. The processing data rate of the last generation FPGA totally fulfills the data rate requested by the NIDS, while the reconfigurability of FPGA allows modifying the inspected rule set upgrading the bitstream of the FPGA. In fact, the set of rules that an IDS has to analyze can rapidly change, depending on the discovery of new viruses or new other types of attacks. In this paper we propose an novel approach that exploit the characteristic of FPGA to efficiently manage the context state swapping. To improve the efficiency of the context state swapping we propose to store in an external RAM only the state of the less active flows. Instead, the most used flows are managed internally by the FPGA using the architecture described in Section V. The remainder of the paper is organized as follows: in Section 2, we discuss previous works on hardware rule matching. In Section 3 the key concept is presented, while in Section 4 we present the implementation of rule matching engine for single stream. In Section 5, we extend the strung matching engine to multiple flows. Section 6 presents the implementation results and finally, Section 7 concludes the paper.

2. RELATED WORKS

Several studies report hardware implementations of pattern matching engines. Pattern matching can be performed implementing in hardware different algorithms used by software version of IDSs like Aho-Corasik[17] [14], or Wu-Manber[3]. Moscola [5] proposed a Deterministic Finite Automata(DFA), supporting multi bytes comparisons and

This work has been partially supported by the European Commission in the frame of the Project FP7-ICT PRISM, contract number 215350.

partial matches. Sidhu and Prasanna[13] mapped a Non-deterministic Finite Automata (NFA) on an FPGA. The use of Bloom Filters [15] leads, if the system can support false positives, to fast and area-saving implementations [19] [16]. Tools for organizing patterns in tries, taking advantage of some prefix sharing rule and thus saving FPGA's area[4][18], have been proposed too. Some works used memories like SRAM, Ram blocks or CAM[6][8], that offer great space for a large number of patterns but that may reduce considerably the speed rate. Other configurations use hybrid hardware-software architectures, exploiting the hardware speed only for partial/approximate string matching[19] and addressing a small portion of traffic to software for further analysis[3]. Moreover an hybrid approach can lead to more complex architectures that may be quickly reconfigured or that can take different actions based on the results of pattern matching[7]. All these works are focused on a single data flow. The actual use of all these systems in a network environment requires the inspected flows to be previously reassembled and sent one at a time to the IDS.

3. KEY CONCEPT

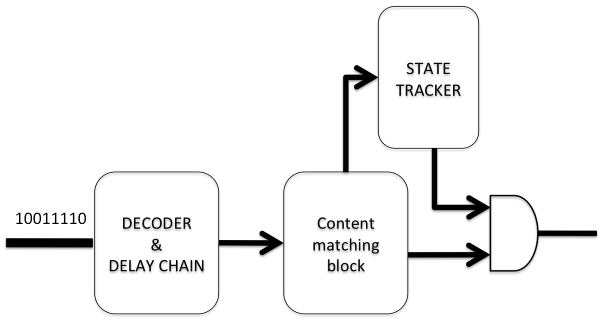

Fig. 1. Basic implementation of a multi-string matching block

Every clock cycle one byte of the packet enters the Decoder and Delay chain as shown in fig. 1. This block provides the inputs to the content matching blocks. These blocks perform the matching of the contents contained in the IDS rules. The results obtained by the content matching blocks are stored in the state tracker block. This block reveals when a full rule is matched. ONLY IF the logical AND between the content matching block (asserting that a content of a specific rule is matched) and the state tracker block outputs (asserting that the other contents of the same rule are matched) is set, THEN we are able to say that the whole rule is matched. In the rest of this paper we will assume that a TCP-Stream Reassembly provides to our framework ordered packets, taking care of all the problems linked to out of sequence and/or overlapping packets. In literature different works that realize an TCP reassembly modules over FPGA are present. Necker [10] proposed an hardware implementation of TCP reassembly and state tracking able to face at most 30 flows. Li [9] implemented a system that reassembles 40 TCP connections using Dual port Ram, reaching up to 2.7Gbps. Finally, note that the need for TCP-Stream Reassembly is a quite restrictive requirement for our architecture. In fact, our framework is able to properly performs the packet inspection when the packets belonging to a flow are reordered. The difference between reassembling and reordering is that, in the first case, the former requires to receive always a certain amount of IP packets in order to form an upper layer data frame (TCP/UDP) before sending it to the rule matching engine, while the latter is able to deliver a packet immediately as soon as the consecutive one is received. This means that the amount of memory required by the reassembler is always greater than the one required for reordering.

4. FPGA IMPLEMENTATION OF RULE MATCHING ENGINE FOR SINGLE STREAM

We present two implementation of the rule matching engine, very similar to the ones discussed in [20]. In the first one, presented in Fig. 2, the input byte enters in a flip-flop chain. The longest content that has to be matched provides the maximum length of the flip-flop chain. In this way the last M entered byte are stored in the flip-flop chain and can be evaluated in parallel. The evaluation is performed by the combinatorial network (shown in the dashed box of Fig. 2). For each content the combinatorial network check if each single character correspond to the expected one and perform the logical AND of all the founded characters.

With respect to [20], we extend the content matching framework to a complex rule matching infrastructure, able to handle multiple contents and rule modifiers. It is, in fact, sufficient to add a global counter, whose task is to count the byte of the analyzed packet, and local registers to store the content of the global counter when the content of a specific rule is found. Moreover one or more flip-flops keep trace of the previously matched contents that compose a complex rule. These flip-flops and registers form the state tracker block of Fig. 2. Suppose that we are interested in matching the content def, we have to check if the third character is equal to $'d'$, **and** the second is equal to $'e'$ **and** the first is equal to $'f'$. Assume that we are interested in matching a rule saying that the content def is placed 20 bytes after the content abc. When the content def has been found, the flip-flop that keeps trace of this event is triggered and the value of the global counter is stored in one of the local registers. The matching of the above mentioned entire rule is obtained as a logical AND of the output of the matching framework asserting that the string def is found, together with a circuit that verifies the distance between the two con-

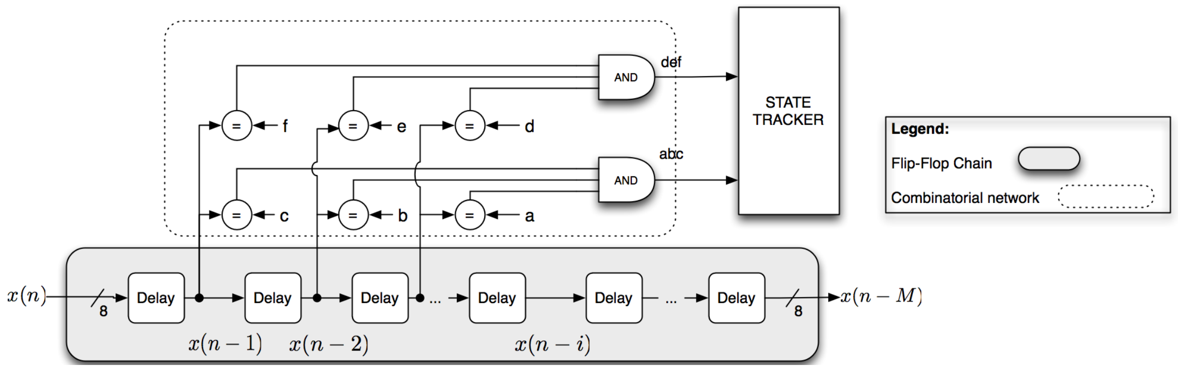

Fig. 2. Basic implementation of a multi-string matching block

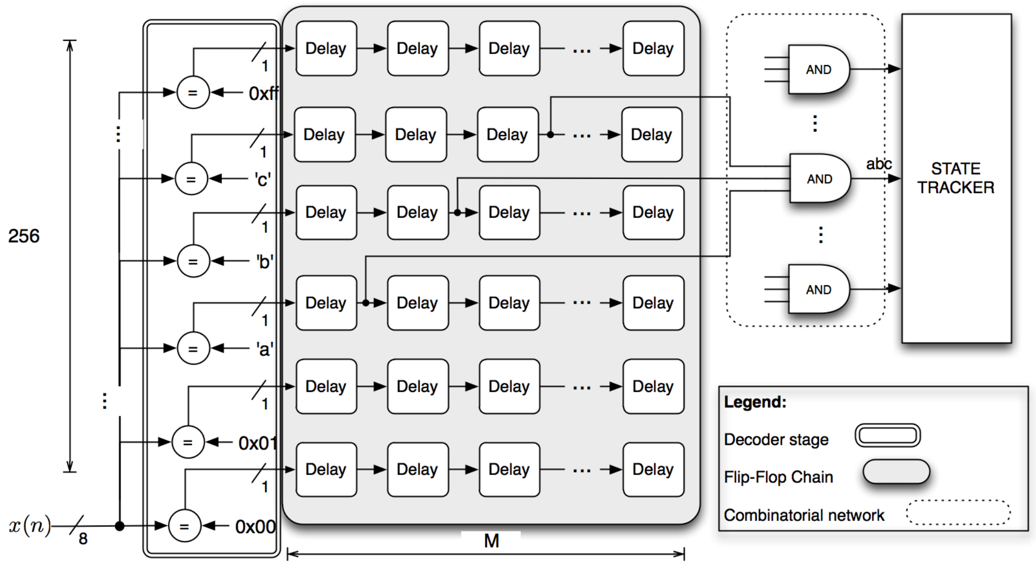

Fig. 3. Implementation of a multi-string matching block with decoded input delay chain

tents and the flip-flop output keeping trace of the matched abc content. The verification of the distance between the two contents is performed with a subtractor operating over the global counter and the local register.

The second architecture differs from the previous one because it shares part of the combinatorial network in order to decrease the resources occupation. The reader can see from Fig. 2 that is possible that the equal operation is performed for each character of each content. When the number of content grows, the number of these comparisons grows too. Therefore it is possible to share resources to perform these comparisons by using a decoding block as presented in Fig. 3. One byte every clock period is entered a decoding block, (see Fig. 3), triggering the flip-flop chain that keeps trace of the correspondence of the entered byte with one of the 256 possible ASCII symbols. In order to match a single string with this architecture it is sufficient to logically AND all the outputs of the flip-flop chains (that track the presence of the symbols composing the content to match) in the desired order. Suppose that we want to match the content EDD, we have to logically AND the output of the third and the second flip-flop in chain that track the symbol D and the first flip-flop of the chain that keeps trace of the symbol E. In Section VI we will show that this architecture use a huge number of flip-flops with respect to the previous one, but allows an important saving in terms of LUTs occupation.

5. EXTENSION OF RULE MATCHING ENGINE TO MULTIPLE STREAM

The architectures presented in previous section performs rule matching, one flow once at a time. In a real network environment, however, a rule matching engine should cope with a sequence of packets belonging to different data streams. Each time a new packet arrives, the engine should check which data stream the packet belongs to, in order to perform the analysis of the right stream, taking into account the state tracker related to the selected flow. Basically the state of a flow is composed by the last characters inside the delay chain and the matching context state of the rule. If a malicious content $abcd$ is split in two packets the end of the first packet contains ab, while the beginning of the second packet contains cd. To detect the whole $abcd$ content, even if packets of a different stream are inserted between these two packets, the characters yet present in the delay chain at the end of a packet must be stored. Moreover, the storage elements of the state tracker must take into account if other contents of the same rule has been matched and the registers used to check the distance between different matches. The method we propose to store these information is composed of two parts:

1. the storage elements of the architecture presented in Fig. 2, $i.e.$ the flip-flops of the delay chain and the registers of the state tracker, are substituted by the multi-input memory element depicted in Fig. 4.
2. a "Least Recently Used" policy is implemented in order to decide a correspondence between the active data flows and the multi-input rule matching engine.

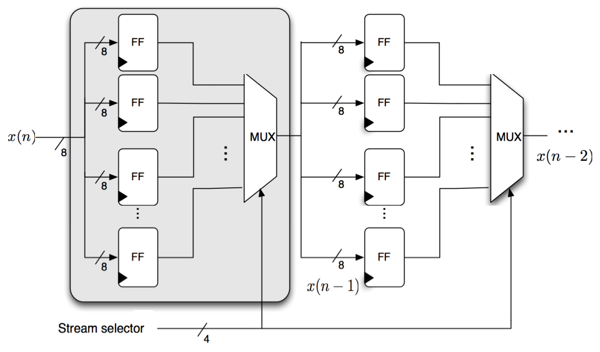

Fig. 4. Implementation of multi-input for the encoded input delay chain

The Stream selector's signal selects one of the flip-flop outputs as the output of the multi-input delay element. Instead, at each clock cycle only one of the 16 Flip-flop is enabled to latch the input coming from the previous stage. The behavior of this structure in totally equivalent to chain of 16X8 RAMs in which the stream selector signal works as the address for the 16X8 RAM. The motivation for using this structure is strictly related to the FPGA organization. In fact, the LUTs of the modern FPGAs can be configured as 16X1 RAMs. Therefore a structure like the one presented in Fig. 4 requires only 8 LUTs for each stage. In Section 6 we will show the impact of this choice in terms of FPGA resources occupation. The structure of Fig. 4 allows swapping between one of 16 different flows in one clock cycle. In fact, the four signals composing the stream selector bus allow swapping between 16 different content. Suppose the previous situation in which the content $abcd$ is split in two packets of a data stream. When the first packet goes into the rule matching engine one of the 16 lines is reserved to the data stream this packet belongs to. When this packet ends, and a new packet belonging to a different stream arrives, the stream selector bus is changed. In this way the overall status of the previous stream is frozen until the occurrence of another packet belonging to the same previous stream. To manage more than 16 streams we have to use additional memory for storing the state of active data flows. The use of additional memory requires to extract the information stored inside the delay chain and the data stored in the state tracker block. Because the length of the chain is more than 100 characters and the state tracker must take into account the trace of thousands of rules, the overall amount of data that has to be transferred (each time the additional memory is used) is in the order of magnitude of several thousands. Supposing a 64 data width memory, the transfers would require tens of clock cycles. During the transfers the rule matching engine is unable to perform its normal operation, therefore each access to the external memory corresponds to waste of computational time and to decrease of the engine throughput. Obviously, also the engine state restoring operation requires access to the external memory and so implies further time loss. The discussed one is very similar to the behavior of computers with multitasking operating systems, in which the CPU registers, representing the machine state, are stored and restored to an external memory at each context swapping. Following the computer analogy, the most frequently used data are usually stored in a cache, in order to minimize the accesses to the external memory. A similar approach is necessary to decide which of the various input streams must be placed into one of the 16 lines of the rule matching engine, and which of them must be stored in the additional memory. We propose to use a "Least Recently Used" algorithm to select which is the stream to be offloaded when a packet of a new stream arrives. For each line we use a counter that increments each time a new packet arrives. If the packet belongs to a stream already present in any line of the engine, the corresponding counter is reset, instead if the packet belongs to a stream not yet present in the engine the stream with the higher counter is offloaded. In this way in the engine always stores the most frequently active streams, minimizing the additional memory access. An additional flag for each engine line is used to signal if a stream keeps the line busy. With the end of a stream this flag is reset so that, when a new stream arrives and there is a free line, that line is reserved to the newly arrived stream and the flag is set again. The use of additional memory for storing informations related to different streams is mandatory if we want to cope with rule matching analysis involving different stream transmitted in more than a packet. The combined use of multi-stream delay elements and additional memory management with LRU policy reduces the time loss. The number of parallel managed streams can be increased placing different chains in parallel at the cost of increasing the FPGA resources occupation. Moreover, the use of Virtex V FPGAs that holds 6-inputs LUTs as basic logic element allows using 64 parallel streams.

6. IMPLEMENTATION RESULTS

Table 1. Synthesis results for the different implementations of single stream engine - Virtex II

		200 rules	400 rules	800 rules
Basic (Fig. 2)	# of Flip Flops	508	1063	1302
	# of LUTs	1676	4301	6506
	# of Slices (utilization [%])	908 (3%)	2315 (9%)	3459 (14%)
With decoder stage (Fig. 3)	# of Flip Flops	1749	4371	4726
	# of LUTs	783	1780	3419
	# of Slices (utilization [%])	769 (3%)	1847 (7%)	2618 (11%)
Hybrid	# of Flip Flops	1656	4073	4361
	# of LUTs	821	2255	4059
	# of Slices (utilization [%])	793 (3%)	1943 (8%)	2740 (11%)

Our rule matching engine has been implemented by using the Xilinx Virtex II Pro XC2V50 FPGA and the Xilinx Virtex V XC5VLX110T FPGAs. All the synthesis has been carried out using the Xilinx XST software [22], imposing a maximum frequency of 125 Mhz as time constraint. This constraint allows sustaining a 1Gbps traffic analysis. The single stream engines, as described in Section 4 has been synthesized for three sets of rules, corresponding to 200, 400 and 800 http rules extracted from the Snort ruleset. For each set of rule we implement three different architectures: one described in Fig. 2, one described in Fig. 3 and an hybrid one. The hybrid architecture uses a partial decoder stage (see Fig. 3) in which only the most frequent characters are decoded. For our purpose we decode only the ASCII codes corresponding to alphanumeric characters. The results for the three architectures are reported in Table 1. The results presented confirm the analysis discussed in Sections 4 and 5. The basic architecture, without decoded stage, requires the highest number of LUTs and the lowest number of Flip-Flop. Instead, the sharing of the decoding operation performed in the architecture of Fig. 3 allows savings around 50% of used LUT, but with an high cost in terms of Flip-Flops. The hybrid architecture is in the middle between the other two. It requires less Flip-Flop than the second architecture, keeping a saving in terms of LUTs with respect to the first architecture. In the single stream rule matching architectures our target is the minimization of the Slices, and therefore the second and the third architectures are the right choices. Instead, if we want to implement the rule matching engine for multiple streams we need to minimize the number of flip-flops and therefore the basic architecture is the most suitable choice. We present in Table 2 results obtained implementing the same rule sets for a 16 concurrent streams rule matching engine. The table shows an overhead that is around only the 30-40% of the resource of the better single stream implementation (the one of Fig. 3). This is due to the massive use of 4-inputs LUTs of the Xilinx Virtex II-Pro as 16X1 RAMs. The number of LUTs composing the combinatorial network is quite similar to the one of the single stream matching engine, as expected. The number of 16X1 RAM block is fixed by the longest content under analysis length, that in our rule sets are 37, 72, and 88, respectively for the 200, 400 and 800 rule sets. Now, we report the results obtained from synthesis on the Xilinx Virtex V XC5VLX110T device. This FPGA uses 6-inputs LUTs, therefore a 64 concurrent stream implementation is a natural choice. We synthesize the single stream architectures with initial decoder stage (Fig. 3) and we compare this architec-

Table 2. Synthesis results for the basic implementation (Fig. 2) of the 16 stream engine- Virtex II

	200 rules	400 rules	800 rules
# of Flip Flops	79	102	239
# of LUTs used as logic	1449	3379	5925
# of RAM16X8	37	72	88
# of LUTs used as RAM	304	576	704
# of Slices # (utilization [%])	914 (3,8%)	2054 (8,7%)	3439 (14,5%)

Table 3. Synthesis results comparison: single stream engine $vs.$ 64 streams engine - Virtex V

	200 rules	400 rules	800 rules
Single Stream			
# of Flip Flops	1566	3630	3914
# of LUTs used as logic	430	1054	2196
# of LUT Flip Flop pairs (utilization [%])	1010 (1,4%)	2428 (3,5%)	4002 (5,7%)
Multiple Stream			
# of Flip Flops	79	100	246
# of LUTs used as logic	968	2258	3733
# of LUT Flip Flop pairs (utilization [%])	1294 (1,8%)	2879 (4,1%)	4527 (6,5%)

ture with the implementation of the 64 concurrent streams rule matching: the data are reported in in Table 3. Similarly to the Virtex II case, the number of 64X1 RAM block is fixed by the longest content length under analysis, as expected. The 64 streams engine has an overhead between 10% and 20% with respect to the best implementation of the single stream engine. In this case is even more evident that the extension to multiple stream functionalities by using the Virtex V FPGA has a low cost with respect to the resource occupation of the overall system.

7. CONCLUSIONS

This paper discusses how to extend an FPGA based rule matching engine to a multi-flow content. The architectural modifications and the collateral schedule method proposed allow the inspection on multiple streams delivered in the same network. Our architecture can switch from the analysis of a stream to another one without time loss. Instead, the LRU based policy minimize the use of external memories needed to store and restore the state of partially inspected flows that exceed the 16 (or 64) concurrent streams. Different rule matching engine architectures have been presented and discussed, in order to identify the right choice to minimize the FPGA resources occupation. In particular, with reference to the Virtex V implementation, the synthesis results show that the extension to a 64 stream engine has a very limited cost in terms of resources occupation.

8. REFERENCES

[1] Sourcefire, “Snort: The Open Source Network Intrusion Detection System” http://www.snort.org, 2003.

[2] M. Almgren, E. Jonsson, and U. Lindqvist, ”A Comparison of Alternative Audit Sources for Web Server Attack Detection”, in Proceedings of the 12th Nordic Workshop on Secure IT Systems (NordSec 2007), Reykjavik University, Oct. 11-12, 2007

[3] R. Proudfoot, K. Kent, E. Aubanel, N. Chen, ”Flexible Software-Hardware Network Intrusion Detection System,” The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, RSP08, pp.182-188, 2008

[4] Z. K. Baker, V. K. Prasanna, ”Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs,” IEEE Trans. on Dep. and Sec. Comp., vol. 3, no. 4, pp. 289-300, Oct.-Dec. 2006

[5] J. Moscola, J. Lockwood, R.P. Loui, and M. Pachos, “Implementation of a Content-Scanning Module for an Internet Firewall,” Proc. 11th Ann. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM ’03), pp. 31-38, 2003.

[6] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, “Granidt: Towards Gigabit Rate Network Intrusion Detection,” Proc. 13th Ann. ACM/SIGDA Int’l Conf. Field-Programmable Logic and Applications (FPL ’03), pp. 404-413, 2003

[7] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos and S. Vassiliadis, ”Packet Pre-filtering for Network Intrusion Detection”, in 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 183-192, San Jose, CA, USA, December 2006.

[8] 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. Napa Valley, CA, April 2004

[9] S. Li, J. Torresen and O. Soraasen, ”Improving a Network Security System by Reconfigurable Hardware”, In proc. of 22nd Norchip Conference, November 2004, Oslo, Norway.

[10] M. Necker, D. Contis, David Schimmel, ”TCP-Stream Reassembly and State Tracking in Hardware”, Proc. of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’02)

[11] A. K. Tummala, P. Patel, ”Distributed IDS using Reconfigurable Hardware,” IPDPS, pp.426, 2007 IEEE International Parallel and Distributed Processing Symposium, 2007

[12] C. Dan Lo, Y. Tai, K. Psarris, and W. Hwang, ”Super Fast Hardware String Matching,” 2006 IEEE International Conference on Field Programmable Technology, Dec. 2006.

[13] R. Sidhu and V.K. Prasanna, “Fast Regular Expression Matching Using FPGAs,” Proc. Ninth IEEE Symp. Field-Programmable Custom Computing Machines (FCCM), 2001

[14] Y. Liu, D. Xu, D. Liu, L. Sun “A Fast and Configurable Pattern Matching Hardware Architecture for Intrusion Detection”. WKDD 2009: 614-618

[15] B. Bloom, ”Space/time trade-offs in hash coding with allowable errors”, ACM, 13(7):422-426, May 1970

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

[17] A.V. Aho, M.J. Corasick,”Efficient String Matching: An Aid to Bibliographic Search”, Communications of ACM: June 1975 Vol. 18 n. 6

[18] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, “Specialized Hardware for Deep Network Packet Filtering,” in 12th Conference on Field Programmable Logic and Applications. Montpellier, France: Springer-Verlag, Sept. 2002, pp. 452 461

[19] Yoshioka, A. Shaikot, Min Sik Kim, ”Rule Hashing for Efficient Packet Classification in Network Intrusion Detection”, Proceedings of 17th International Conference on Computer Communications and Networks, 2008. ICCCN ’08.

[20] I. Sourdis, D. N. Pnevmatikatos, S. Vassiliadis, ”Scalable Multigigabit Pattern Matching for Packet Inspection”, IEEE Trans. VLSI Syst. 16(2): 156-166 (2008)

[21] R. Smith, C. Estan, S. Jha, ”XFA: Faster signature matching with extended automata”, IEEE Symposium on Security and Privacy (Oakland), May 2008

[22] XST User Guide, available at http://toolbox.xilinx.com/ docsan/xilinx5/pdf/docs/xst/xst.pdf