2010 International Conference on Field Programmable Logic and Applications

Exploiting Dynamic Reconfiguration for FPGA based Network Intrusion Detection Systems

Salvatore Pontarelli, Claudio Greco, Enrico Nobile, Simone Teofili, Giuseppe Bianchi
Consorzio Nazionale InterUniversitario per le Telecomunicazioni (CNIT)
University of Rome "Tor Vergata", Via del Politecnico 1, 00133, Rome, ITALY

***Abstract*—A Network Intrusion Detection System (NIDS) inspects the traffic flowing in a network to detect malicious content such as spam, viruses, and so on. Hardware based solutions appear necessary to face the performance requirements emerging when the goal is to deploy such systems in high speed network scenarios. However, the appropriate choice of the hardware platform is believed to be subject to at least two requirements, usually considered independent each other: i) it needs to be reprogrammable, in order to update the intrusion detection rules each time a new threat arises, and ii) it must be capable of containing the typically very large set of rules of existing NIDSs. The goal of this paper is to show that reprogrammability can be further exploited to reduce the resource requirements for the chosen platform. Specifically, we propose an FPGA-based solution that classifies and dispatches traffic to elastic buffers, connecting one buffer at a time to a dynamically reconfigurable rule matching core. This core supports only the appropriate subset of detection rules. A worst-case analysis shows that the saving in hardware resources is achieved with a relatively small buffer space, currently available in cheap, low end, FPGA boards, with no impairment on the resulting throughput.**

I. Introduction

The demand for network security and protection against threats and attacks is ever increasing, due to the widespread diffusion of network connectivity and the higher risks brought about by a new generation of Internet threats. Network Intrusion Detection Systems (NIDS) play a key role in such a scenario. A NIDS is a system that analyzes the traffic crossing the network, classifies packets according to header and further inspects payload information with respect to content-matching rules for detecting the occurrence of anomalies or attacks.

NIDS deployment has been boosted by the emergence of open source systems, such as Snort [1]. Its open source software nature, in conjunction with its flexible rule-based language and the availability of a very active users' and developers' community, has fostered the deployment of NIDS systems in network scenarios where cost/benefits trade-offs might not justify the adoption of commercial, high end, dedicated systems.

However, the ever increasing deployment of higher link rates and the ever growing Internet traffic volume appears to challenge NIDS solutions purely based on software. Especially, payload inspection (also known as deep packet inspection) appears to be very demanding in terms of processing power, and calls for dedicated hardware systems either devised to implement a specific set of IDS functions [**?**], or to act as IDS pre-filter for offloading a remote software-based IDS [2], [3]. The approach presented in [2], [3] combines the flexibility of IDS software and the processing power of a hardware implementation. These systems perform a first processing of the packet traveling into the network and, if a suspicious string has been found, the packet is forwarded to a PC running a software based NIDS. In this way only a little fraction of the traffic is analyzed by the software allowing using these systems also in networks with high data rates. The hardware IDS, instead, implements only a simplified version of the rules that must be checked. The simplified rule version is used to identify only the small subset of the whole traffic that potentially can carry the complete rule.

Any attempt to foster a smooth and viable migration towards hardware-based solutions, meanwhile retaining the success and easy of adoption of software based ones, must challenge at least two specific requirements.

First, envisioned hardware-based NIDS solutions must be extremely flexible in their support of new emerging rules devised to handle new threats or attacks. This does not seem to be a particularly challenging requirement, in sight of the availability of FPGA-based systems, which intrinsic reconfiguration ability allows to update the inspected rule set by simply upgrading the FPGA bitstream.

Second, hardware-based NIDS systems must be operated on low cost widely deployed commodity platforms. However, the large number of rules (see section II) that must be implemented in hardware sets forth tight requirements on the amount of logic resources. Resorting to "more capable" FPGA boards is not free of concerns, not only because of cost reasons (high end FPGA boards are probably still well affordable in most practical deployment scenarios), but rather for critical mass reasons. For instance, the majority of the networking research community appears having adopted a specific board, namely, NetFPGA [4], for development purposes, in sight of the large amount of network-specific pre-developed functionalities, the abundant documented use-cases, the frequently dedicated events, and, most important, the large user community base. But, the limited resources

1946-1488/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.13

offered by such a board make hard (to the best of our knowledge) to deploy the complete set of rules of an IDS such as Snort.

A. Our contribution

The idea in this paper tries to exploit FPGA dynamic reconfiguration to save FPGA logic resources. A dynamically reconfigurable FPGA is a device that can be partially reconfigured while the rest of the application running in the not reconfigured part can continue to operate without interruptions. Many works propose to use his features for different aims: in [5], [6], the target is the reduction of used resources, while in [7], [8] is used for design of high reliable circuits.

All proposed hardware implemented IDS systems [2], [3] inspect all the incoming traffic with respect to a static set of rules defined at compile time. A rule describes the characteristics that a packet must satisfy in order to be identified as a malicious packet. Matching a rule requires both the classification of the application protocol ($e.g.$ http, TCP, mail, executable files, etc.) and the subsequent identification of one or more suspicious contents ($e.g.$ strings, bytes, regular expressions), placed in specific positions inside the data flow. We can note, however, that it is necessary to perform different and specific types of analysis for packets transporting different types of application protocol and data. Therefore, is unnecessary to search inside the payload all the suspicious contents, but only the content related to the application protocol (usually identified exploiting the TCP/UDP ports) transported by a packet. As a straightforward consequence, the easiest way to classify the packets traveling on the internet is a classification based on the type of transport protocol, (TCP or UDP) and on the port ($e.g.$ port 80 for http traffic, port 21 for ftp transfer and so on). In Section II will be further discussed how to divide the rule set of a NIDS with respect to the different types of application protocol. Now we only remark that while the first classification (port and type of transport protocol) does not require many computational resources, the subsequent rule matching task is very resource demanding.

To exploit the dynamic reconfiguration capabilities of FPGAs, we divide the rule set of the NIDS hardware implementation into complementary sets that are activated by different types of traffic. After, we can deploy different FPGA configurations corresponding to the different complementary sets of rules. The packets related to a specific set of rules are analyzed by configuring the FPGA with the corresponding configuration bitstream. But, each FPGA reconfiguration requires a certain amount of time, that corresponds to an off-line interval, and the available time to analyze a packet is very little compared with the FPGA reconfiguration time. Therefore, the reconfiguration is unfeasible if applied for each packet. To overcome this problem we propose classify each packet in a first stage of the FPGA, after we store the packets in different queues, depending on the type of data transported, and to analyze by a reconfigurable second stage of the FPGA all the packets coming from the same queue. In this way the off-line time needed to reconfigure the FPGA is payed only each time we decide to swap between different queues. The system we propose therefore is composed by two parts: a static part that performs the first classification, and a dynamic part. The static part dispatches traffic to elastic buffers, connecting one buffer at a time to the dynamic part. In the dynamic part are continuously loaded different configuration in order to inspect different type of traffics. The partial reconfiguration can be implemented maintaining the static part fully functional (see [13]). In this way no packets are discarded during the reconfiguration phases because the static part continues to store in the elastic buffers the incoming traffic. The proposed approach allows saving hardware resource, sharing the same resources of the dynamic part between different configuration for inspection of different type of traffic. The penalty of this type of approach is the need of memory for storing the packets in the queues. Our method allows to perform a trade-off between the amount of FPGA logic resources and the amount of requested memory. However, we remark that the proposed system has been targeted to be implemented on a specific card [4] that is widely used in networking community and that fulfills the above mentioned requirements: the amount of hardware resources available with the FPGA equipped in that board is limited and the board is equipped with a 64 MBytes DRAM usable for implementing the queues. Finally we note that, due to the latency introduced by the queues, this system is suitable for intrusion detection systems, not for intrusion prevention systems that use automatic active reactions. Currently, the reaction to attack detected in a network usually requires an human operation. In fact, the unwanted countermeasures due to automatic intervention activated by false positive attack detection causes an unacceptable degradation of the quality of service in the network. In this scenario the latency of our proposed method is completely sustainable. The rest of the paper is organized as follows: in section II the description of the typical NIDS rules is presented, discussing how complementary sets can be created. In section III the proposed strategy is presented, while in section IV the evaluation in terms of throughput and queue sizing are presented and finally, in section V the conclusions are drawn.

II. Description of Snort rules

In this section we present a description of NIDS rules, following the rule structure used for Snort [1]. The Snort IDS performs deep packet inspection of the incoming flows checking if at least a rule is matched by one of the inspected flow and eventually produceing an alert containing the information that allows identifying the malicious flow. Due to the very different kinds of inspection that can be performed

(different protocols, virus and exploits for various operating systems and so on) the rules can be defined in a very flexible way in order to define one or more sets of bytes within a flow that allow the unambiguous identification of every anomaly. The rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rules action and the header information that must be compared with the packet, like: protocol, source and destination IP addresses and net masks, and the source and destination ports information. The rule option section contains alert messages and information which parts of the packet should be inspected to determine if the rule action should be taken. Snort performs several types of content matching searching for content, uricontent (searches the normalized request URI field), regular expression. Moreover, sometimes an anomalous behavior is identified when a content, common in the payload, is matched in a specific position in the packet (for example in the first 4 bytes of the payload). Snort exploits a set of rule modifiers (i.e. depth, within, offset,distance) to identify the position where a content has to be matched to produce an alert. The complexity of the Snort rules implies that their hardware implementation is difficult. Even resolving the problem related to the flow reassembly task, the realization of modifiers, PCRE and the other Snort keyword could require a big effort and a huge amount of hardware resources. Readers interested in hardware realization of PCRE can for example refer to [9], [10], [11]. Therefore we develop an approximate rule matching hardware engine by using only a subset of each rule. If we accept to delete some element from a rule we obtain some false positive alert, but do not produce false negative. To simplify the hardware implementation of the rule matching engine, for each rule is extracted only one of the content composing the rule. This choice corresponds to a big simplification of the rules, but introduces a number of false positive alerts. These false positive increases the amount of traffic forwarded to the software IDS. However, as reported in [2], the use of such method allows to filter up to 90% of incoming traffic. Table I provides the distribution of the number of snort rules divided by the protocol and the port. The rules of the first row must be applied for all the ports of TCP protocol, while the rules of the second row are used only for HTTP traffic. Instead, in the third row we collected all the TCP rules that use ports different from the HTTP port. The fourth and fifth rows are rules for UDP and ICP protocols, while the last row is related to rules for IP protocol, that must be applied for all TCP and UDP packets.

From the data in table I we can easily identify two complementary sets. The set called A is used when a packet containing HTTP traffic must be scanned. The A set is composed by the rules related to HTTP, and by the rules (TCP and IP) that must be checked independently from the port. The B set is composed by the TCP rules that have a port different from the HTTP port, and by the TCP and IP rules. These two set are not completely disjoint because some rules should be checked in both the cases of HTTP or not HTTP traffic. Finally, the UDP and ICMP rule set is disjoint from the set of TCP rules, and therefore could be inserted both in the A or in the B set, or in a third set of rules. We can insert the UDP and ICMP rules in the rule set that requires less logic resources, to balance the resource usage.

Table I
NUMBER OF SNORT RULES WITH RESPECT TO PROTOCOL

Protocol	number of rules	set
TCP rules with any port	162	A,B
HTTP rules	2610	A
TCP (non HTTP) rules	3482	B
UDP rules	384	A
ICMP rules	131	A
IP rules	34	A,B

Even if we have developed a very specific framework employing relaxed rules to identify the traffic that has to be sent to Snort, however, the proposed solution (i.e. exploiting the dynamic reconfiguration capabilities of FPGAs in order to divide the rule Snort set), can be employed without modification even if we want to implement in an FPGA the whole set of complete Snort rules.

III. PROPOSED STRATEGY

In this section we describe our strategy to combine dynamic reconfiguration and rule classification to minimize the amount of hardware needed to realize a hardware based NIDS. In fig. 1 we depict the schema of our system.

We use as target technology for the system implementation a reconfigurable FPGA with network interfaces (NetFPGA) [4]. The NetFPGA is a PCI card equipped with a Xilinx FPGA Virtex II Pro, 4 Gigabit Ethernet ports, SRAM and DRAM banks and therefore, even if the FPGA used by this board is obsolete, is a suitable candidate for a fast prototype of our architecture. The use of the NetFPGA allows exploiting the integrated 4 Gigabit Ethernet ports, the DRAM memory banks provided by the PCI card and an interface with a PC host usable both for debugging and for controlling the developed hardware. Finally, The Xilinx FPGA Virtex II Pro can be dynamically reconfigured, as foreseen for our application. All the blocks of the schema in Fig. 1 are contained in the FPGA, with the exception of the queues. These queues are realized with the external DRAM memory, while the control part of the queues is realized inside the FPGA. The main inputs of this system are the incoming packets to be analyzed. These inputs are classified depending on the type of traffic transported by the packet. The classification divides the incoming packets into two flows, corresponding to the two complementary A and B sets. This classification is a static part of the reconfigurable design because must be always performed by the system. After the packets are classified, they enter one of the two

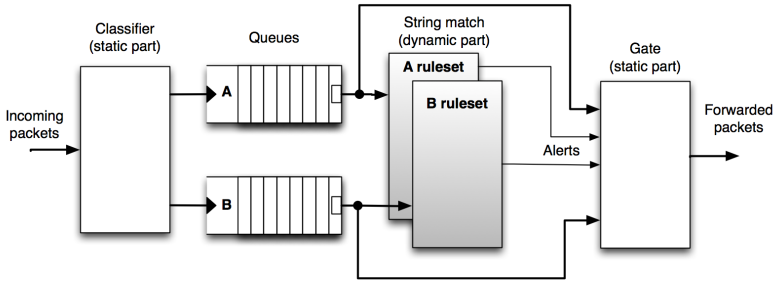

Figure 1. schema of the proposed system

queues depending on the result of classification. The outputs of the queues are provided as inputs to the content matching engines contained in the dynamic reconfigurable part of the design. The queues are written by the classifier and are read by the content matching engines. When the A ruleset is loaded in the FPGA it read the packets stored in the A queue until it is empty, while the B queue accumulates packets. When the A queue is empty, the B ruleset is loaded inside the FPGA, and the system start to process the packets in the B queue. Both the A and B ruleset reconfigurable blocks provide as outputs some alert signals that instruct the gate block to forward the detected packet to the software IDS for further analysis. The blocks realizing the ruleset are the most resource consuming ones, because they must realize the string matching of thousand of rules. The hardware realizing the string matching has been extensively discussed in [12] [3], and is reported in Fig. 2. We briefly describe the structure of this block in order to identify the reconfigurable part of the system.

An 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 bytes are stored in the flip-flop chain and can be evaluated in parallel by the combinatorial network (shown in the dashed box of Fig. 2). For each content, the combinatorial network checks if each single character correspond to the expected one and performs the logical AND of all the founded characters. The switching between two different ruleset corresponds to the modification of the combinatorial part of the string matching engine. Fixing the length of the flip-flop chain as the greater between the length of the longest contents of the two ruleset, the same chain is able to provides the right inputs to both the combinatorial networks realizing the A or the B ruleset. This part represents the reconfigurable part of the system and is also the most resource consuming part of the design, growing with the number and with the complexity of the implemented rules. In Table II we report the synthesis data of the rule matching engine for the A and B rulesets.

The dynamic reconfiguration mechanism we propose follows the method presented in [13]. An area of the FPGA, called partially reconfigurable region (PRR) has bees identified to and will be reserved for implementing one of the complementary ruleset. These ruleset represent the partially reconfigurable module (PRM). Each PRM implementation a single dynamic task that will be mapped into a PRR. From the data of table II we can see that the PRR must be able to host about the 36% of the FPGA, corresponding to the most area demanding ruleset. The other ruleset can be loaded in the same PRR obtaining a saving of 26% of resource with respect to a static implementation in which the FPGA is configured to realize both the A and the B ruleset.

Table II
SYNTHESIS RESULTS FOR THE DIFFERENT IMPLEMENTATIONS OF STRING MATCHING ENGINE - VIRTEX-II PRO 50

	A ruleset	B ruleset
# of Flip Flops	5853	7090
# of LUTs	11134	12837
# of Slices	6373	8882
(utilization [%])	(26%)	(36%)

IV. THROUGHPUT EVALUATION AND DEFINITION OF QUEUE DEPTH

To evaluate the effectiveness of our system we should prove that it is able to work at wire speed. The used NetFPGA is able to collect data by using a Gigabit Ethernet interface. Parallelizing the collected data one byte at time, a 125 MHz clock can be used to drive the classifier depicted in Fig, 1. Instead, the queues must be realized as asynchronous FIFOs, with a write clock of 125MHz (8 ns), and a read clock with a frequency of 133MHz (7,5ns). The read frequency must be greater that the write one because, we must sustain some period in which the data cannot be read because the dynamic part of the system is in the reconfiguration phase. Following the method described in [13] the reconfiguration time of the dynamic part can be evaluated in the order of 10 ms., and during the reconfiguration phase we can suppose that at maximum 1.25 Mbytes ($10ms/8ns$) can enter the queues. If we define Q the maximum number of bytes that can be stored in the queue during the processing phase, the system is able to empty a queue of Q bytes in a time $T = Q \cdot 7,5ns$. During this time, the other queue can accumulate up to $T/8ns$ bytes. Therefore, in the worst case the queue accumulates $Q = T/8ns + 1.25MBytes$.

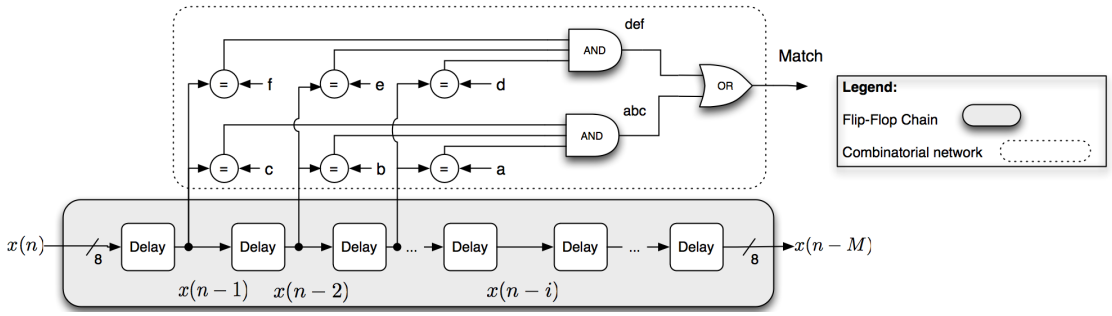

Figure 2. Basic implementation of a rule matching block

The corresponding equation

$$Q = 1.25MB + Q \cdot 7,5ns/8ns \quad (1)$$

has as solution $Q = 20MB$. This amount of data completely fits the queue size that can be implemented in the NetFPGA using the available 64MB DRAM.

V. Conclusions and future works

In this paper the dynamic reconfiguration features of Xilinx FPGA are used to minimize the amount of logic resource needed to realize a Network Intrusion Detection System. The paper shows that our proposed system can sustain the rate of data coming from a Gigabit ethernet interface. Moreover, we evaluate the length of the queues needed to sustain the gigabit rate. Our developed system can be easily implemented on a widely used board for networking application such as the NetFPGA. The presented method is based on a first classification block that split the flow of the incoming packets towards two queues. Each queue is related to a specific block that is dynamically loaded when the data in that queue must be analyzed. The data are accumulated in a queue when the system is analyzing the other queue or when it is in the reconfiguration phase. With a suitable sizing of these queues the presented system is able to work without packet losses. The presented classification is the simplest one, but can divide the ruleset only in two subsets. The presented method can be extended to divide the ruleset in many subsets, increasing the number of dynamic blocks and therefore improving the area saving capabilities.

Acknowledgment

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

References

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

[2] Haoyu Song, T. Sproull, M. Attig, J. Lockwood, “Snort offloader: a reconfigurable hardware NIDS filter”, Field Programmable Logic and Applications, 2005. International Conference on, 24-26 Aug. 2005

[3] C. Greco, E. Nobile, S. Pontarelli, S. Teofili “An FPGA Based Architecture for Complex Rule Matching with Stateful Inspection of Multiple TCP Connections”, SPL - VI Southern Programmable Logic Conference, 24-26 March 2010

[4] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, J. Luo, “NetFPGA-an open platform for gigabit-rate network switching and routing”, IEEE International Conference on Microelectronic Systems Education 2007

[5] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, “A time-multiplexed FPGA”. In Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines, 1997.

[6] Y. Adachi, K. Ishikawa, S. Tsutsumi, and H. Amano, “An implementation of the Rijndael on Async-WASMII”, Int. Conf. on Field-Programmable Technology (FPT), 15-17 Dec. 2003.

[7] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, “Dynamic fault tolerance in FPGAs via partial reconfiguration”, IEEE Symposium on Field-Programmable Custom Computing Machines, 2000 17-19 April 2000.

[8] M. Gokhale, P. Graham, E. Johnson, N. Rollins, and M. Wirthlin, “Dynamic reconfiguration for management of radiation-induced faults in FPGAs”, 18th International Parallel and Distributed Processing Symposium, 26-30 April 2004.

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

[10] C. Lin, C. Huang, C. Jiang, S. Chang, “Optimization of Pattern Matching Circuits for Regular Expression on FPGA”, IEEE Trans. on VLSI Syst., 15(12), 2007

[11] Y.H.E. Yang, W. Jiang, V. Prasanna, “Compact architecture for high-throughput regular expression matching on FPGA”, Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 30–39, 2008

[12] I. Sourdis, N. Dionisios, S. Pnevmatikatos, “Scalable Multi-gigabit Pattern Matching for Packet Inspection”, IEEE Trans. VLSI Syst. 16(2): 156-166 (2008)

[13] P. Lysaght, B. Blodget, J. Mason, J. Young, B. Bridgford, “Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration of Xilinx FPGAS”, Proceedings of the International Conference on Field Programmable Logic and Applications (FPL-06), pages 12–17, 2006