Chapter 23
IDS Rules Adaptation for Packets Pre-filtering in Gbps Line Rates

Simone Teofili, Enrico Nobile, Salvatore Pontarelli and Giuseppe Bianchi

Abstract The enormous growth of network traffic, in conjunction with the need to monitor even larger and more capillary network deployments, poses a significant scalability challenge to the network monitoring process. We believe that a promising way to address this challenge consists in rethinking monitoring tasks as partially performed *inside* the network itself. Indeed, in-network monitoring devices, such as traffic capturing probes, may be instructed to perform intelligent processing and filtering mechanisms, so that the amount of data ultimately delivered to central monitoring entities can be significantly reduced to that strictly necessary for a more careful and fine-grained data inspection. In such a direction, this chapter focuses on the design and implementation of an hardware-based front-end pre-filter for the topmost known Snort Intrusion Detection System (IDS). Motivated by the practical impossibility to pack a large amount of legacy Snort rules over a resource-constrained hardware device, we specifically address the question on how Snort rules should be adapted and simplified so that they can be supported over a commercial, low-end, Field Programmable Gate Array (FPGA) board, meanwhile providing good filtering performance. Focusing on about one thousand Snort rules randomly drawn from the complete rule set, we experimentally determine how these rules can be simplified meanwhile retaining a comparable detection performance with respect to the original, non adapted, rules,

S. Teofili (✉) · E. Nobile · S. Pontarelli · G. Bianchi
Consorzio Nazionale InterUniversitario per le Telecomunicazioni (CNIT)/University of Rome "Tor Vergata", Via del Politecnico 1, 00133, Rome, Italy
e-mail: Simone.Teofil@uniroma2.it

E. Nobile
e-mail: Enrico.Nobile@uniroma2.it

S. Pontarelli
e-mail: Salvatore.Pontarelli@uniroma2.it

G. Bianchi
e-mail: Giuseppe.Bianchi@uniroma2.it

N. Blefari-Melazzi et al. (eds.), *Trustworthy Internet*,
DOI: 10.1007/978-88-470-1818-1_23,

when applied over a “training” dataset composed of a relatively large traffic trace collected from a regional ISP backbone link. We then validate the performance of the adapted rules against additional collected traffic traces. We show that about 1000 adapted Snort rules can be supported over a low-end FPGA based Snort pre-filter, with 93% data reduction efficiency.

Keywords Online traffic analysis · SNORT · IDS · FPGA

23.1 Introduction

The constant growth in diffusion and performance of networks is accompanied to an increase in the number of hacking and intrusion incidents. Consequently Intrusion Detection Systems (IDS) has been proposed to detect the presence of such type of incidents. Most IDSs monitor traffic flows and inspect packet payloads for detecting predetermined attack patterns (signatures). Different kinds of intrusions may be taken into account: shell-codes that exploit Operating Systems vulnerabilities to gain unauthorized access to a host computer, policy violations in the use of a corporate network, port scans, and so on. Moreover, in recent years, IDS rules have been extended to detect also user mis-behavior, such as exchange of pornography material and so on. Obviously, the collection of IDS rules must be promptly updated in order to cope with emerging threats or monitoring needs, and this yields a number of rules that is constantly increasing both in cardinality as well as in rule complexity.

Open source, widespread deployed, Network Intrusion Detection System (NIDS) such as Snort [1] are software based. Because of software limitations, as well as limited traffic capturing capabilities of ordinary PC network cards, they are mostly used in relatively low loaded small networks, whereas their exploitation for backbone links is questionable, due to the huge traffic volume involved and the multi gbps line rates.

Hardware based NIDS have been proposed to face these limitations, and sustain traffic inspection at wire speed [2]. These systems are installed over traffic probes, and act as pre-filter. Their goal is to detect the subset of possible malicious streams, and reduce the amount of traffic data delivered to a back-end software NIDS. The effectiveness of such systems can be measured in terms of data reduction capabilities: a significant data reduction would bring about the possibility to reuse legacy software-based IDS and permit cheap deployments.

One on the best candidate technologies for the development of such hardware NIDS systems are the FPGA. These reprogrammable components are designed to be configured by the customer or designer after manufacturing, therefore they are called “field-programmable”. This devices can be programmed to accomplished the pattern matching activities needed by the NIDS at very high speed. Moreover, due to their reprogrammability, the set of rules that are checked can be easily updated downloading a new bitstream in the FPGA.

Designing an hardware-accelerated IDS pre-filter over these devices, is however a non trivial task. Indeed, IDS rules such as the Snort rules set, are not limited to "basic" string matching. Rather, they may include multiple types of matching (i.e., content matching, byte matching, uri matching), as well as they may require the matching of multiple contents further regulated by "modifiers", indicating the position in the flow in which the content must be located, or the distance between a content and the previous. Finally, some rules may require the matching of regular expressions. Even if several works [3, 4] have described a thorough FPGA implementation of complex rules, the amount of logic resources and the design effort needed to implement these rules on the FPGA can be overwhelming, especially when the goal is to move away from proof-of-concept implementations supporting a few tens of rules, and reach the practical real-world target of supporting an order of a thousand rules or more.

A practical solution to this issue consists in devising an IDS pre-filter which supports a set of *loosened* rules. The basic idea is very simple, and can be easily understood over the following trivial example. Assume that a Snort rule R_{orig} is triggered when two content patterns C_1 and C_2, for instance separated by a modifier, are matched. Consider now a new rule, R_{adapt}, devised to match only C_1. Its implementation would obviously require a lower amount of hardware resources. Moreover, its detection capability would be a *superset* of that of the original rule, and would not incur in *false negatives*, i.e., all cases detected by the original rule would also be detected by the new, "adapted", rule. The disadvantage of such rule adaptation is that the filtering performance clearly decreases, as more streams will be detected and delivered to the monitoring back-end for further inspections (false positives).

A trade-off emerges between filtering performance and ability to "pack" rules in the hardware front-end pre-filter. According to our in-field experience acquired during the experimental assessment work described in Sect. 23.6, there is no "obvious" adaptation mechanism which appears to optimize such a trade-off. For instance, if Snort rules were adapted so that only one single content were matched over the pre-filter, hardware implementation would be very efficient and simple, but the false positives rate may become unacceptably large. And to make things worse, filtering performance largely varies, and strongly depends on *individual* Snort rules.

To face these issue, we have resorted on an experimentally-driven, heuristic, rule adaptation methodology, made possible by our availability to access real traffic data delivered over the backbone of a regional ISP. Specifically, we have collected a number of large traffic traces, and used one of them as "training" set. We have then iteratively compared the number of alerts detected by a legacy Snort software, implementing a set of *original* Snort rules, with that detected by a pre-filter implementing *adapted* rules. At each iteration, we have identified which individual adapted Snort rules were the main cause of false positives (and why), and modified these rules accordingly. Performance assessment was then carried out by testing the adapted rule set over different network traces (up to 68 GB traffic).

The rest of the chapter is organized as follows. Section 23.2 provides the necessary background on Snort rules. Section 23.3 discusses the related work on

Table 23.1 Description of keywords modifiers

Modifier	Description
Offset: *N*	The search for the content begin after *N* characters
Depth: *N*	The search for the content ends after *N* characters
Distance: *N*	The distance between two contents is at least *N* characters
Within: *N*	The distance between two contents is less than *N* characters

hardware IDS filters. Section 23.4 presents our FPGA hardware implementation of the Snort pre-filter front-end. Section 23.5 details the Snort rules adaptation methodology, taking also into account how to adapt the rules so as the resulting hardware implementation is simplified. Section 23.6 presents experimental results performed over real traffic traces. Finally, conclusions are drawn in Sect. 23.7.

23.2 Description of Snort Rules

The Snort IDS performs deep packet inspection on incoming flows, checking whether some specific rules are matched, in which case an action is performed (i.e., alert, log or pass). Rules are divided into two logical sections: *rule header*, including packet header field information (source/destination IP, protocol, source/destination port), and *rule options*, containing alert messages, information on the patterns to match including their position inside the flow, etc. Moreover some keywords related to the state of the session and the direction of the packets (i.e., from or to the server) can be present. Almost all the Snort rules include one or more among the following keywords: content, modifiers, uri-content, and PCREs.

Content specifies a fixed pattern to be searched in the packets payload of a flow. If the pattern is contained anywhere within the packets payload, the test is successful and the other rule options tests are performed. A content keyword pattern may be composed of a mix of text and binary data. Most rules have multiple contents.

Modifiers identify a location in which a content is searched inside the payload. This location can be absolute (defined with respect to the start of the flow) or relative to the matching of a previous content. Snort modifiers are listed and described in Table 23.1.

Uricontent searches the normalized request URI field in the packet.

PCRE (*Perl Compatible Regular Expression*) define regular expression pattern matching using the PERL syntax and semantics. Even if PCREs give high flexibility to the description of the content to be searched for, the hardware implementation of a generic PCRE may require a lot of logic resources [3, 5]. In the Snort rules set, PCRE are usually used to describe some particular kind of attack. For instance, the following rule describes a buffer overflow:

```
alert tcp EXTERNAL_NET any- > HOME_NET 1655 (msg:''EXPLOIT
ebola USER overflow attempt''; flow:to_server, established;
content:''USER''; nocase; pcre:''/^USER\s [^\n]{49}/smi'';)
```

The PCRE search for a string starting with "USER", followed by a space (the escape sequence \s), followed by a maximum of 49 characters that differ from the newline character. If the rule is matched the IDS detects a too long user name that corresponds to a tentative buffer overflow.

The flexibility of the Snort rules corresponds to a difficulty in its hardware implementation. As will be discussed in the next section, the implementation of modifiers, of PCRE and of the other keywords could require a big effort and a huge amount of hardware resources.

23.3 Review of FPGA Based IDS Systems

In this section a brief overview of FPGA based IDS systems is presented. The aim of the section is to identify some common issues of these hardware implementation that we propose to solve by our rule adaptation procedure.

Packet pre-filtering has been originally proposed in [2]. This work exploits Bloom filters implemented on FPGA to detect malicious contents inside the inspected packets. The main limit of this approach is related of the inflexibility of this string matching approach.

More flexible implementations of pattern matching can be obtained by using Deterministic Finite Automata (DFA) [6] or Nondeterministic Finite Automata (NFA) [7]. These solutions can be applied to pattern matching and extended to matching regular expression such as PCRE [3, 4, 7]. Implementations based on DFA or NFA rapidly grow in size when the number of rules to be checked increase, and when Snort modifiers or PCRE are considered. Indeed, a DFA for a regular expression of length L can have $O(2^L)$ states [8]. NFA avoids state explosion but requires a more resource consuming hardware for its realization [7]. As such, DFA or NFA based pre-filter implementations challenging to support thousands of Snort rules may be very costly.

Our developed architecture is based on an alternative approach presented in [9], called shift-and-compare. It can give better results in the FPGA implementation, compared to DFA/NFA ones, since the basic logic block of an FPGA can be configured as a shift-register, reducing the amount of resources needed to implement this architecture. In particular, the shift-and-compare architecture allows sharing a significant part of memory elements between all the rules, thus enabling the implementation of thousand of rules on the same FPGA. Moreover, by using suitable registers, as we will show in details in Sect. 23.4, is possible to extend the shift-and-compare architecture for complex rule matching.

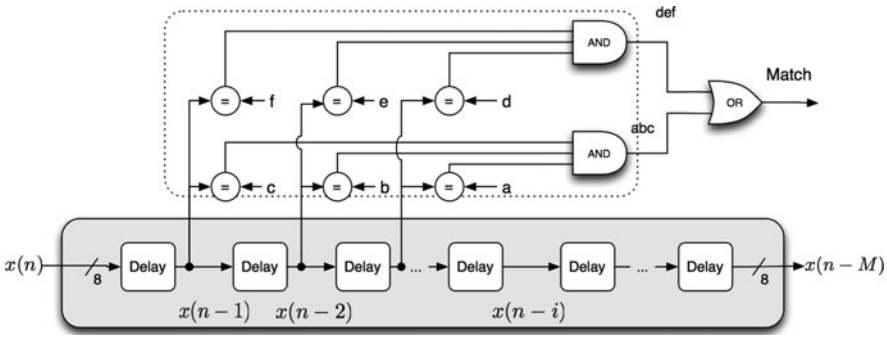

Fig. 23.1 Basic implementation of a multi-string matching block

Summarizing, the problem related to the current available FPGA based IDS systems are:

- Content matching alone is insufficient: in several cases, rule modifiers must be already accounted for by the FPGA pre-filter, in order to avoid a large amount of false positives
- The amount of logic resources grows linearly with the number of contents to be matched
- The most consuming logic resources are the memory elements representing the state of partially matched complex rules.

23.4 Snort Pre-filter Hardware Architecture

Our Snort pre-filter has been implemented over a Xilinx FPGA Virtex II Pro. FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow to connect together the logic blocks to perform more complex functions. Logic blocks are implemented by programmable Look-up Tables (LUT), that can be configured to perform combinational functions, like AND and XOR and also include memory elements, which may be simple flip-flops or more complete blocks of memory. Both the configuration of the logic blocks and of the interconnection are stored in a binary file called bitstream. The Virtex II pro FPGA has been integrated in a low-end PCI card equipped with an FPGA, 4 Gigabit Ethernet ports, SRAM and DRAM banks, called NetFPGA [10]. Even if the FPGA used by this board is obsolete, it is a suitable candidate for fast prototyping.

The content matching hardware illustrated in Fig. 23.1 follows the approach presented in [11]. Input bytes enter in a flip-flop chain. The longest content to be matched sets the maximum length of the flip-flop chain. The last M entered bytes are stored in the chain and can be evaluated in parallel. The evaluation is

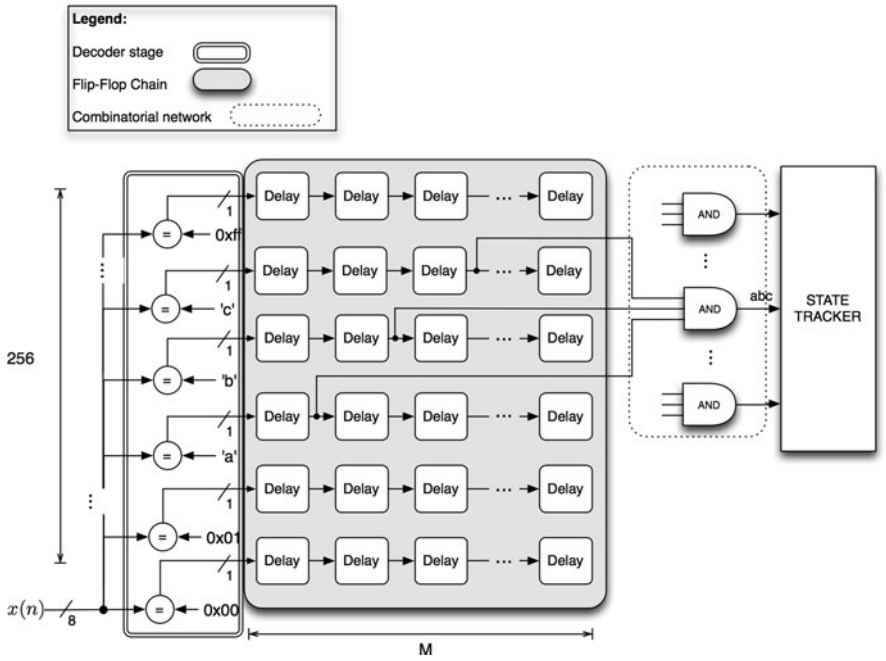

Fig. 23.2 Implementation of a multi-string matching block with decoded input

performed by the combinatorial circuit (shown in the dashed box of Fig. 23.1). For each content, the combinatorial network checks if each single character corresponds to the expected one and performs the logical AND of all the found characters. For instance, suppose that we are interested in matching the content *def*. We check if the third character is equal to '*d*', and the second is equal to '*e*' and the first is equal to '*f*'. If all these matches occur, a signal called *match* is raised.

When the number of rules increases, also the number of comparators, and therefore the resource occupation of the combinatorial circuit, increases. This extra complexity negatively affects the maximum operating frequency of the multi-string matching block. To overcome such a limitation, the bytes' comparators can be shared between the different string to be matched, as presented in [12], where the implementation of a multi-string matching block with decoded input delay chain is presented. The circuit implementing the decoded multi-string matching block is reported in Fig. 23.2.

The number of flip-flop is greatly increased, while the number of logic resources is decreased. But, as we already noticed before, the FPGA is able to optimal pack the chain of shift-registers in logic blocks (a LUT can be configures as a chain of 16 flip-flops), achieving only a limited resource occupation when long chains of shift registers are used. When the number of strings to be matched increases, this decoding circuit gives better results both in terms of area occupation

Table 23.2 Synthesis results for the different implementations of multi-string matching circuits

		200 rules	400 rules	800 rules
Basic (Fig. 23.1)	# of flip flops	508	1063	1302
	# of LUTs	1676	4301	6506
	# of slices (utilization [%])	908 (3%)	2315 (9%)	3459(14%)
With decoder stage (Fig. 23.2)	# of flip flops	1749	4371	4726
	# of LUTs	783	1780	3419
	# of slices (utilization [%])	769 (3%)	1847 (7%)	2618 (11%)

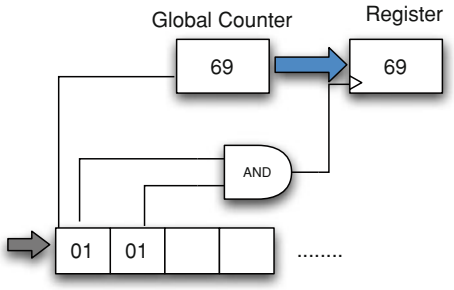

Fig. 23.3 First content matching

and achieved frequencies. The results in terms of resource occupation (LUT and FF) of such two implementations are reported in Table 23.2.

Our rule matching engine has been implemented by using the Xilinx Virtex II Pro XC2V50 FPGA available on the NetFPGA [10]. The multi-string matching circuits has been synthesized for three sets of rules, corresponding to 200, 400 and 800 rules extracted from the Snort ruleset. The results presented confirm that 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 allows savings around 50% of used LUT, but with an high cost in terms of Flip-Flops.

We have extended this basic content matching architecture with the capability to support more complex rule matching including Snort modifiers and simple PCREs. A global counter and dedicated registers were added to track partial matches. Figures 23.3 and 23.4 show an example where the rule to be matched includes a modifier "within":

```
content:''|01 01|''; content=''|00 00 02|''; within 10;
```

In this example, two binary contents |01 01| and |00 00 02| must be at a distance less than 10 bytes. The first part of this rule, i.e. the match of the content |01 01|, is performed by the two inputs AND gate. When the content is matched the value of the global counter is stored in a register (i.e. Fig. 23.3). Now, when the second content is matched, the system also checks if the difference between the global counter and the value stored in the register is less than ten bytes (i.e., Fig. 23.4). This extension is resource consuming because a register and a comparator must be instantiated for each part in which the rule is decomposed.

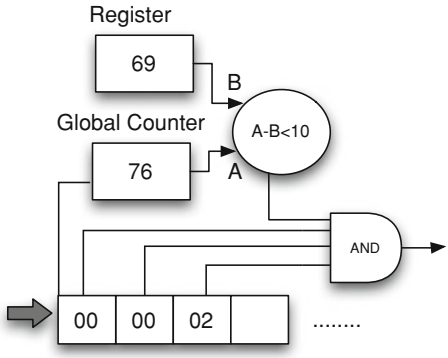

Fig. 23.4 Second content matching and modifier check

Table 23.3 Resource occupation of the string matching engine

Feature	
RAMB	0 out of 232 (0%)
Slices	6684 out of 23616 (28%)
Maximum frequency	125 MHz

Table 23.3 reports the hardware resource occupation after the synthesis of one thousand rules adapted as described in Sect. 23.5 Only 28% of the total FPGA logic resources were used. Note that the rest of the FPGA logic in the proposed framework is used to implement the modules necessary to manage the packets transmission or reception, instantiate Ethernet interfaces and debug operations as reported in [10].

23.5 Rule Adaptation

A major goal of our work consisted in the identification of how to adapt Snort rules for obtaining an efficient pre-filter hardware implementation meanwhile retaining a limited amount of false positives. In details, a crucial aspect is the identification of which "part" of the rules is most effective in detecting the specific attack. At least in principle, this analysis should be carried out at the level of *individual* rules, as, to the best of our knowledge, no general adaptation guidelines appear applicable, and indeed (as it will become clear in the rest of this section) a same approach applied to different rules provided widely different false positive performance.

We operated through the analysis of the set of Snort rules available in the Snort public distribution (5709 rules). We iteratively operated as follows.

First, we extracted from every rule the longest one among the content or uri-content included in each rule (in most generality, a rule specifies more than one matching). Note that this first step is somewhat analogous to what done in literature works in which only one content for rule is employed [2, 13]. The use of only one content per rule would permit an extremely efficient hardware implementation,

as it would allow to get rid of all the logic devised to cope with multiple contents, hence including registers and the comparators needed to track which contents have been previously matched for each rule. As an example, the following legacy Snort rule:

```
alert tcp, HOME_NET any -> $EXTERNAL_NET HTTP_PORTS
(msg:''SPYWARE-PUT Hijacker coolwebsearch.cameup runtime
detection''; flow: to_server, established; uricontent:
''svc=''; nocase; uricontent:''lang=''; nocase; uricontent:
''type=''; nocase; uricontent: ''mode=''; nocase; uricon-
tent:''art=''; nocase; uricontent:''acct=''; nocase; uri-
content: ''url=''; nocase;uricontent:''category=''; nocase;
uricontent: ''view=''; nocase; sid:6242; rev:2;)
```

was relaxed exploiting a single content[1]:

```
content:''category=''; nocase;
```

The list of so-relaxed rules (containing only the longest content/uri-content) was then tested against a "training" 7 GB trace containing a 40 min real network traffic captured over the backbone of a regional ISP. The resulting several thousand alerts signaled by Snort were then collected and re-attributed to each relaxed rule. In all cases, alerts were generated because the selection of the longest content resulted into a very common (or very short) string. For example, a content like "User-Agent:" is the longest one in many rules, but at the same time is a content present in all HTTP requests. The results of this automated analysis provided a set of long common contents[2] that cannot be used to represent a Snort rule. These contents were inserted into a so-called "blacklist". The described longest content extraction process was iteratively repeated, excluding blacklisted contents, (i.e., for a rule containing one of such contents the second longest one was chosen at the second iteration, and so on. The next step consisted in identifying all the rules that, after the content iteration described above, either included only a small content (2 or 3 bytes), or did not include any remaining content at all. For these rules, we selected via trials and errors (tested over the captured data trace) the modifier or simple PRCE starting from the simpler ones (from the point of view of an hardware implementation) available in the considered rule. For instance, in rule:

```
alert tcp $SMTP_SERVERS 465 -> $EXTERNAL_NET any (msg:''
SMTP SSLv2 Server_Hello request''; flow:from_server,
established; content:''|04|''; depth:1; offset:2; con-
tent:''|00 02|''; depth:2; offset:5; sid:3497; rev:4;)
```

the content was clearly not sufficient for an efficient matching. Indeed, the longest content (00 02) resulted in practical matches for almost all flows on port 465. The rule was therefore relaxed by exploiting the modifiers "depth" and "offset":

[1] To simplify implementation, the uricontent keyword was relaxed into a content keyword.

[2] The specific list being: {User-Agent, Server, Agent, Internet, Connection, complete/search?, /index.php.}

Table 23.4 Result for rule adaptation

	Number of packets	Number of flow
Original rule set	1534	511
Longest content only	86979	9745
Proposed adaptation	2831	857

```
content:''|00 02|''; depth:2; offset:5;}
```

Similarly, another example is rule:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 110 (msg:''POP3
USER overflow attempt''; flow:to_server,established;
content:''USER''; nocase; isdataat:50, relative; pcre:''/
^USER\s [^\n]{50,}/smi''; sid: 1866; rev:12;)
```

where the content "USER" alone cause a huge number of false positives; its relaxed version exploits the PCRE, i.e.:

IDS Rules adaptation for packets pre-filtering in gbps line rates

```
pcre'':/^USER\s [^\n]{50,}/smi''}
```

Unfortunately, this final step necessary to identify which part of the Snort rule is more effective, requires a work that only partially can be automated. The rules with the most representative part causing an excessive number of false positive have to be manually analyzed to understand how they can be best approximated.

The results obtained by our Snort rules adaptation process are shown in Table 23.4. The first row reports the packets and TCP flows identified as malicious by using the original snort ruleset. The second row presents the result by applying only the longest content of the rule. It can be seen that this approach is inapplicable because the number of false positive is excessive. The third row finally shows the results obtained by the final set of adapted rules.

23.6 Experimental Result

This section presents results obtained by testing the adapted rule set over a real traffic scenario. For simplicity, we have randomly chosen a subset of 1000 Snort rules for hardware implementation over the Snort pre-filter. Experiments have been performed on ten captured traffic traces, amounting to a total of 68 GB of traffic.

In order to assess the effectiveness of the pre-filter, we have compared the following scenarios:

- Unfiltered trace: all the traffic has been directly fed to the Snort software supporting the 1000 rules in their non-adapted (full)version.
- Filtered trace: the traffic has been first delivered to the Snort hardware pre-filter, supporting the 1000 reduced Snort rules. The output of the pre-filter has been then delivered to the Snort software, acting in this configuration as the back-end monitoring application, supporting the same set of rules in their full version.

The first performance metric consists in measuring the ability of the Snort pre-filter to reduce the data delivered to the back-end application. Of the 68 GB traffic, only 4.83 GB of data was delivered; in other words, the Snort pre-filter was able to

Table 23.5 Experimental Result

	Unfiltered trace	Filtered trace
Alert	3579	3540
False negative	0	190
False positive	0	151

filter out 93% of the original traffic. Note that this result was obtained by using a pre-filter implementation which does not restrict to capture only the packets matching a given (relaxed) signature, but further delivers all the *subsequent* packets in the relevant flow.[3]

The second performance metric involves the assessment of the pre-filter effectiveness in terms of false positives and false negatives. Table 23.5 compares the results obtained in the two scenarios. The Snort software operating over the unfiltered traces has revealed 3579 alerts while it has revealed only 3540 alerts over the filtered traces.

The relatively large amount of false negatives obtained (190 undetected attacks over the 3579 ones, according to the unfiltered trace results) is readily explained and imputable to two relatively marginal technical issues to date still affecting our current prototype implementation.

The first issue involves the coding of special characters used in uri-content matching. Our current pre-filter implementation does not yet support the translation of Unicode characters to ASCII which are provided in Snort by the http_inspect preprocessor (whose implementation is out of the scope of our work). As a result, our pre-filter does not match the Unicode representation of the character "/". This clearly emerges, for instance, with the rule identified by the sid 895 (message WEB-CGI redirect access), which produced 97 false negatives since its relaxed version was expected to match the uri-content "/redirect", i.e., including the special character '/'.

The second reason for false negatives is imputable to issues which are not related to our Snort pre-filter, but appear to be related with specific requirements of the legacy Snort *software* in correctly detecting and parsing TCP flows. Indeed, in order to correctly parse a flow, the Snort software requires not only to receive the specific packets matching a specific signature, but it also requires some extra information on the TCP flow itself. To face this issue, we have implemented a software pre-processing module running on the back-end system before the actual Snort legacy software, and devised to "forge" TCP ACK packets and the TCP three way handshake which are filtered out by the HW pre-filter. Our module appears effective in most of the cases, as it provides a

[3] Pre-filter architecture details are out of the scope of this chapter. But, in brief, a filtering table was added to the Snort pre-filter. The table was automatically updated with a flow key extracted from a matching packet, and managed using an LRU (Least Recently Used) policy. All packets whose flow matched an entry of the filtering table were then forwarded. This permits to feed the Snort application operating in the back-end with multiple packets belonging to a same matching flows, and not only isolated matching packets.

formally valid input to the Snort software, and particularly the information needed by the Snort software for determining the state of the session (i.e., if the session is established or not) and the direction of the packets (i.e., from or to server). However, the in-depth analysis of the actual false negatives reveals that even if the relevant signature were correctly captured by the pre-filter and delivered, in some cases, the extra information (ACKs and TCP handshakes) forged by the software pre-processing module were insufficient to drive the Snort legacy software to correctly parse the captured flow.

Finally, note that besides the false negatives detected because of the two above discussed implementation issues, we would have also expected false negatives to occur because of signatures spreading across multiple packets. Indeed, our pre-filter does not perform packet reassembly as this would require to keep per-flow states, and therefore it is vulnerable to signatures split across multiple packets. However, in practice, no such cases have been identified, also because we have employed relaxed rules which are much shorter than the exact rules they were derived from.

Concerning false positives, we have found only 151 cases. A closer scrutiny revealed that 134 out of 151 false positives were due to a single rule, namely sid 8428 (i.e. openssl get shared ciphers overflow attempt), which was undetected in the unfiltered scenario because, in the original trace, the TCP three-way handshake associated to these alerts were not included in the captured traces (i.e., the relevant TCP handshake had happened before the trace capture starting time), whereas in our case the relevant handshakes were properly forged by our software pre-processing module. In other words, these 134 cases should be more properly interpreted as false negatives of the Snort legacy software because of boundary effects of the considered packet trace, rather than false positives of the Snort pre-filter). This leaves an excellent performance of just 17 "true" false positives for our Snort pre-filter operating on normal real world traffic.

23.7 Conclusions

We presented the design and implementation of an hardware-based front-end pre-filter for the topmost known Snort Intrusion Detection System (IDS). Moreover, we have proposed an experimentally-driven heuristic rule adaptation methodology of the Snort rules so that they can be supported over a commercial, low-end, FPGA board, meanwhile providing good filtering performance. In particular, leveraging a large amount of real world traffic traces, we have determined through a cycle of experiments, how these rules can be properly simplified. Finally we have demonstrated that about 1000 adapted Snort rules can be supported over a low-end NetFPGA hardware based Snort pre-filter, with 93% data reduction efficiency, retaining a comparable detection performance with respect to the original, non adapted, rules.

References

1. Sourcefire: Snort: The open source network intrusion detection system. http://www.snort.org (2003)
2. Haoyu Song Sproull, T., Attig, M., Lockwood, J.: Snort offloader: a reconfigurable hardware NIDS filter. In: International Conference on Field Programmable Logic and Applications (2005)
3. Yang, Y.H.E., Jiang, W., Prasanna, V.K.: Compact architecture for high-throughput regular expression matching on FPGA. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 30–39 (2008)
4. Bispo, J., Sourdis, I., Cardoso, J., Vassiliadis, S.: "Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues", Reconfigurable Computing: Architectures, Tools and Applications. Springer
5. Lin, C., Huang, C., Jiang, C., Chang, S.: Optimization of pattern matching circuits for regular expression on FPGA. IEEE Trans. VLSI Syst. **15**(2), 1303–1310 (2007)
6. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of 11th Annual IEEE Symposium Field-Programmable Custom Computing Machines (FCCM '03), pp. 31–38 (2003)
7. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using FPGAs. In: Proceedings of Ninth IEEE Symposium Field-Programmable Custom Computing Machines (FCCM) (2001)
8. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. ACM SIGCOMM Comput. Commun. Rev. **38**(4), 207–218 (2008)
9. Baker, Z.K., Prasanna, V.K.: Automatic synthesis of efficient intrusion detection systems on FPGAs. IEEE Trans. Dependable Secur. Comput. **3**(4), 289–300 (2006)
10. Lockwood, J., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., Luo J.: NetFPGA-an open platform for gigabit-rate network switching and routing. In: IEEE International Conference on Microelectronic Systems Education (2007)
11. Sourdis, I., Dionisios, N., Pnevmatikatos, S.: Scalable multigigabit pattern matching for packet inspection. IEEE Trans. VLSI Syst. **16**(2), 156–166 (2008)
12. Greco, C., Nobile, E., Pontarelli, S., Teofili, S.: An FPGA based architecture for complex rule matching with stateful inspection of multiple TCP connections. Programmable Logic Conference (SPL), 2010 VI Southern, pp. 119–124, 24–26 March 2010
13. Sourdis, I., Dimopoulos, V., Pnevmatikatos, D., Vassiliadis, S.: Packet pre-filtering for network intrusion detection. In: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (2006)