Packet Pre-filtering for Network Intrusion Detection

Ioannis Sourdis,* Vasilis Dimopoulos,♯ Dionisios Pnevmatikatos,♯‡ Stamatis Vassiliadis*

*Computer Engineering Lab.
Electrical Engineering Dept,
TU Delft,
The Netherlands
{sourdis, stamatis}@ce.et.tudellft.nl

♯ Microprocessor and Hardware Laboratory
Electronic and Computer Engineering Dept,
Technical University of Crete,
Chania, Crete, Greece
{dimopoulos, pnevmati}@mhl.tuc.gr

‡ Institute of Computer Science (ICS),
Foundation for Research and Technology - Hellas (FORTH),
FORTH-ICS, member of HiPEAC
Heraklion, Crete, Greece
pnevmati@ics.forth.gr

ABSTRACT

As Intrusion Detection Systems (IDS) utilize more complex syntax to efficiently describe complex attacks, their processing requirements increase rapidly. Hardware and, even more, software platforms face difficulties in keeping up with the computationally intensive IDS tasks, and face overheads that can substantially diminish performance.

In this paper we introduce a packet pre-filtering approach as a means to resolve, or at least alleviate, the increasing needs of current and future intrusion detection systems. We observe that it is very rare for a single incoming packet to fully or partially match more than a few tens of IDS rules. We capitalize on this observation selecting a small portion from each IDS rule to be matched in the pre-filtering step. The result of this partial match is a small subset of rules that are candidates for a full match. Given this pruned set of rules that can apply to a packet, a second-stage, full-match engine can sustain higher throughput.

We use `DefCon` traces and recent `Snort` IDS rule-set, and show that matching the header and up to an 8-character prefix for each payload rule on each incoming packet can determine that on average 1.8 rules *may* apply on each packet, while the maximum number of rules to be checked across all packets is 32. Effectively, packet pre-filtering prevents matching at least 99% of the SNORT rules per packet and as a result minimizes processing and improves the scalability of the system. We also propose and evaluate the cost and performance of a reconfigurable architecture that uses multiple processing engines in order to exploit the benefits of pre-filtering.

*This work was partially supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

TU Delft and FORTH-ICS are member of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'06, December 3–5, 2006, San Jose, California, USA.
Copyright 2006 ACM 1-59593-580-0/06/0012 ...$5.00.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: Network Monitoring

General Terms

Design, Security.

Keywords

Intrusion Detection, Packet Inspection, Packet Pre-filtering, Reconfigurable Computing.

1. INTRODUCTION

High speed and always-on network access is commonplace around the world, creating a demand for increased network security. Intrusion Detection Systems (IDS) such as Snort [12] are currently the most efficient solution for network security. Instead of only checking the header of each incoming packet, IDS also scan the payload of the packets to detect suspicious contents. These systems must be able to process thousands rules per incoming packet and require update mechanisms to renew their rule-set with new descriptions of known attacks. This required flexibility and the fast processing rates are a good match for reconfigurable technology, rather than for general purpose processors, exploiting specialized circuitry and parallelism.

In the past years, many researchers have worked on reconfigurable IDS focusing mostly on the payload scan, which turns out to be the most computationally intensive task: Fisk and Varghese report that payload scanning accounts from 31% up to 80% of Snort's execution time, the larger

value corresponding to web-intensive traffic [6]. Several techniques for reconfigurable IDS static pattern and regular expression matching have been proposed [3, 4, 7, 9–11, 15, 16]. Additionally, efficient packet classification techniques such as [13] are also required for the header matching part of the IDS.

A packet classifier and a content matching engine used to be efficient enough for low-speed networks when implementing the detection core of an IDS that comes after packet reassembly and reordering. However, networks becoming faster and IDS systems, such as Snort, are becoming more complex supporting more efficient attack descriptions; therefore a simple merging of the packet classification and the content matching is not enough to detect hazardous packets. More precisely, Snort IDS rules include statements which, for example, define payload regions where specific patterns should be matched (`depth, offset`) or require a pattern to be matched within a number of bytes after matching another pattern (`within, distance`). Consequently, matching each rule requires a *separate* specialized module to keep track of the payload matches and detect which parts of the payload are valid. Even though, in software each rule might be matched separately (presumably sequentially), in hardware it used to be the case that all payload patterns were matched in parallel and then a simple AND with the packet classifier outcome would produce a rule match. However, requiring a separate module per rule to implement these new IDS syntax features, is not scalable and introduces significant overheads.

In this paper we introduce a packet pre-filtering approach as means to alleviate the above overheads and improve IDS scalability in terms of area cost and performance. More precisely, header matching (a 5-tuple filter i.e. Source IP Address, Destination IP Address, Protocol, Source Port and Destination Port) and a relatively low-cost pattern matching module (matching 2-10 bytes per rule) can filter out the majority of the Snort rules and point out a small subset to be fully matched.

Our pre-filtering approach is based on the observation that a single incoming packet usually will not match (even partially) many attack descriptions. Especially, when part of the payload is included in the filter, it is unlikely that a packet matches multiple payload patterns for several rules. Not excluding other options, our solution could be integrated in a full-featured IDS detection engine as follows: the pre-filtering module determines a "candidate matching" rule subset per packet, and then (possibly multiple) specialized processing engines are employed to fully match these rules. This approach exploits parallelism between the match of different rules, that is not restricted to our proposed reconfigurable architecture; the exact rule processing can also be assigned to multiple threads on the same or multiple processing cores.

Whenever the pre-filtering modules output a rule ID then a specialized module is reserved to match the rule and afterwards released (either at the end of the packet or due to a match or mismatch before the end of the packet). In the rare occasion, where there is no available specialized engine to match a rule, the packet should be reported with the indication that was not fully examined and then policies defined by the user could be applied. We provide experimental evidence suggesting that our proposal is promising. In this paper:

- We introduce the packet pre-filtering approach to determine the few rules (out of thousands) that could possibly match per incoming packet.
- We discuss techniques to integrate the packet prefiltering module into a full-featured hardware or software IDS detection system.
- We introduce a new priority encoder design which is pipelined and therefore scales in terms of performance as the number of inputs increases. In addition, the priority encoder reports sequentially *all* the active inputs, based on a statically defined priority.
- We use DefCon11 traces and Snort v2.4 ruleset to show that in the worst case 32 out of about 3,200 rules are detected for payload match per packet, while the average number of rules that need to be checked for payload match is another order of magnitude smaller, requiring minimal processing.
- In addition, we provide implementation results of a reconfigurable packet pre-filtering module and preliminary results of a complete reconfigurable IDS detection core.

The remainder of the paper is organized as follows: In section 2, we discuss the motivation of our proposal, providing details about Snort syntax features which make IDS more complex. In section 3, we present our packet pre-filtering approach, while in section 4 we describe ways to integrate the pre-filtering module with hardware and software intrusion detection systems. In Section 5 we present simulation results showing the effectiveness of our solution, and implementation results of a reconfigurable packet pre-filtering module. Finally, in Section 6 we conclude the paper.

2. SNORT IDS

Before describing our approach in detail, we first discuss the Snort syntax features which create fundamental difficulties in IDS implementation and motivate our packet pre-filtering design. Table 1 depicts some of the Snort syntax features which make the IDS rules more complicated. The above commands change the original meaning of the payload content Snort rule parts (either static patterns or regular expressions) adding extra constraints regarding the placement of the matching patterns in the packet payload. Consequently, rules might specify the packet payload part where a pattern should be matched, relative either to the beginning or the end of a packet or relative to a previously matched pattern. In addition, commands such as `byte_test` and `byte_jump` select and test a byte payload field using several numerical or logical operators. Each Snort rule might specify different payload constraints, such as the ones mentioned above, to describe a suspicious packet. Therefore, each rule would possibly require a separate FSM-like module to keep track of the satisfied conditions, specify the parts of the payload which are valid for each pattern to match, and store payload byte fields to be tested using the `byte_test` and `byte_jump` commands. In software, the above features might not effect the entire implementation of the detection engine, since the matching is presumably sequential, even though more processing is required. On the other hand, in

Table 1: Current SNORT syntax features which make IDS tasks more computationally intensive.

Feature	Description
`depth`	specifies how far into a packet Snort should search for the specified pattern.
`offset`	specifies where to start searching for a pattern within a packet.
`distance`	specifies how far into a packet Snort should ignore before starting to search for the specified pattern relative to the end of a previous pattern match.
`within`	makes sure that at most N bytes are between pattern matches.
`isdataat`	Verify that the payload has data at a specific location, optionally looking if data relative to the end of the previous content match.
`byte_test`	test a byte field against a specific value (with operator i.e. less than (<), greater than (>), equal (=), not (!), bitwise AND (&), bitwise OR (ˆ) and various options such as `value, offset, relative, endian, string, and number_type`). Capable if testing binary values or converting representative byte strings to their binary equivalent and testing them.
`byte_jump`	allows rules to be written for length encoded protocols. By having an option the reads the length of the portion of data, then skips that far forward in the packet, rules can be written that skip over specific portions of length-encoded protocols and perform detection in very specific locations. Several options are supported such as `byte_to_convert, offset, relative, multiplier <value>, big/little endian, string, HEX/DEC/OCT, align and from_beginning`
`dsize`	tests the packet payload size.

hardware, where the implementation needs to be in parallel (since performance is critical), these syntax extensions introduce significant cost and might also limit performance.

3. PACKET PREFILTERING

In this section, we present the functionality of our packet pre-filtering approach and design details of our reconfigurable hardware implementation.

In the past, pre-filtering techniques have been applied for static IDS pattern matching. In order to match multiple IDS patterns in software, Markatos *et al.* use a two step approach that detects whether an incoming stream contains *all* the pattern characters (possibly in arbitrary positions) and only then perform a full match of the search pattern [8]. More recently, they proposed Piranha, that extends the first checking step with for 32-bit "rare" substrings that will enable fewer rules to be matched in the second step [1]. Their approach is software-based and proposed to replace the main execution loop of Snort. Baker and Prassana employed a pre-filtering technique for reconfigurable IDS pattern matching. They modified their "shift-and-compare" design, reducing the area cost, adding some uncertainty and allowing false positives, to filter out streams that would not match their pattern set [2].

Most hardware-based techniques suffer from the limitation that they search the payload for all patterns in the *entire* ruleset while ignoring rule headers, i.e. packet header information. In essence, they search for thousands of patterns while the packet header might specify that we are interested in only a few tens or so patterns. If we consider that some patterns are only one or two bytes long, one could easily create packets that cause hundreds of accurate yet inconsequential matches. Even normal traffic packets would often trigger invalid rules. If such pre-filtering methods are used stand-alone, they will often "cry wolf". On the other hand, if used in conjunction with a higher-level software IDS they could be tricked into flooding it with false positive pattern matches, possibly causing it to waste more time "sorting through the garbage" than it would scanning the entire payload.

Software implementations of IDS, such as Snort, can group the rules into different sets based on their rule headers. Specifically, TCP and UDP rules can be grouped based on their source and destination ports, ICMP rules based on their ICMP type and IP rules based on their protocol. This grouping creates sets of rules that are compatible and may be activated at the same time. This fact allows software IDS processing a packet to discover a single rule set that covers a large portion of possible attacks and perform a single payload scan for the patterns in that set. In practice however, to minimize processing IDS software may opt for a quick and not so refined approach; Snort just used the destination port for TCP and UDP rules [14], resulting in larger groups but minimal header processing and classification. This represents a basic tradeoff in software systems between time to process the header and space since more payload test strings must be simultaneously checked.

Our key observation in packet pre-filtering is that matching a small part of each rule's payload combined with matching the header information (Source & Destination IP/Port and Protocol) can substantially reduce the set of the possibly matching rules compared to using only header matching as in previously proposed approaches. Using merely the header description results in multiple applicable IDS rules, which can be up to several hundreds in the case of a Snort-like IDS [5]. However, when adding to the filter a few bytes of payload patterns, this candidate set of IDS rule can be significantly reduced. That is because it would be relatively rare for a single incoming packet to match payload search patterns of multiple rules.

Our IDS packet pre-filtering approach relies on the above conjunctures to minimize the set of rules required to be matched per incoming packet. Figure 1 offers the block diagram of our proposed design. The prefiltering module is designed for reconfigurable hardware and therefore can update its supported IDS ruleset via reconfiguration. The top part of the figure illustrates the overall system arrangement. Incoming packets are first filtered through the packet pre-filtering module (matches the first part of each rule i.e. header plus a few bytes of payload pattern); subsequently, only the remaining part of the candidate rules, reported by the pre-filtering, are fully matched in a separate hardware or software module/sub-system. This way, the flow of the

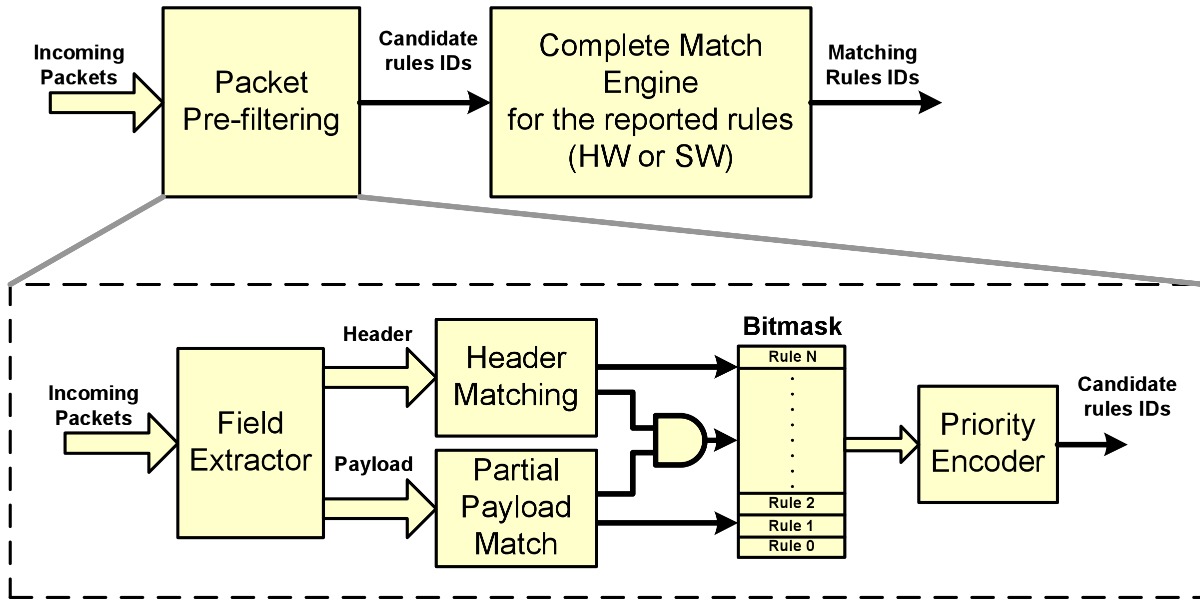


Figure 1: The Packet Pre-filtering block diagram.

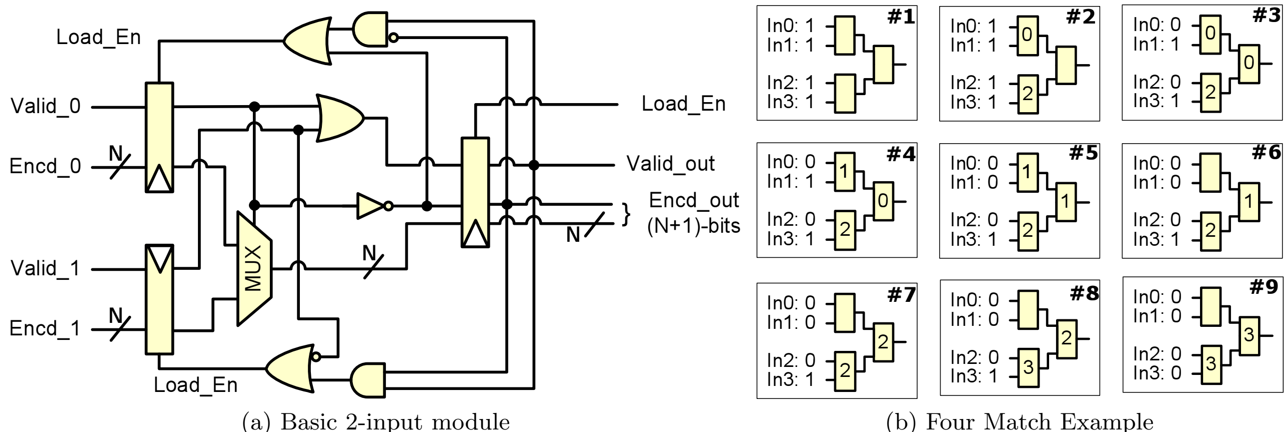
incoming packets is not stalled, since the parts of the rules matched in the prefiltering phase are not matched again.

The bottom part of Figure 1 expands in detail the internals of the pre-filtering block. The incoming packets feed a field extractor module, which performs header delineation, field separation, and payload extraction. The packet header is sent to the Header Matching module that performs the necessary header classification and reports a bitmask of potential matching rules. The payload is sent to the partial Payload Match module which also reports a bitmask of potential matching rules. Depending on each rule's definition, the two bitmasks are combined to provide the logical AND of the two pieces of information (the AND gate in the figure) or can be used directly, i.e. in the case of rules with header constraints but without payload checks or the opposite. The list of possible matches is reported to the full match module using a priority encoder, although in a purely hardware implementation and depending on the implementation of the full match module it could be reported as a full bitmask.

We have described our proposal in terms of a hardware implementation; indeed it seems that it better fits a hardware instead of a software implementation. In software, header matching can be relatively efficient: specific comparisons against fixed-location fields can be performed in a tree-structure and occurs exactly once per packet. However implementing the pre-filtering technique may require scanning the payload part of the packet twice, first for the pre-filtering and once more for the actual match. A hardware implementation overcomes this problem through the use of parallelism; if such parallel resources are available in a software implementation (for example in the form of multi-core general-purpose processor or a network processor), then our pre-filtering approach can be proven efficient.

Header Matching: The header fields enter the packet classification module, which performs a more fine-grained grouping than Snort. For header classification, we use 3 to 5 of all the packet header fields: source and destination IP address and protocol type are used for all rules, with the source and destination ports being additional parameters for TCP/UDP rules and the ICMP type for ICMP rules. Here we have to make two observations: (a) these fields involve the IP header as well as the TCP/UDP headers and the ICMP header, and (b) additional header fields can be used in the Snort rules, but are *not* used for the header classification, so as to avoid excessive number of small groups. The header fields are registered and forwarded to a pipelined comparator module. This module discovers all active rule sets and can also be used to inform the software of the best applicable rule set (thus avoiding the cost of software header matching).

Partial Pattern Matching: Similarly, the packet payload is scanned using partial search patterns. From each Snort rule specifying one or more payload search contents, we select the first pattern and match a constant number of its prefix bytes (between 2 and 10 bytes in our experiments). The first pattern of a rule is selected so that our pre-filtering module will match the first part of each rule. If the pattern is shorter than the selected number of prefix bytes then the full pattern is matched. The static pattern matching is performed utilizing DCAM, a pre-decoding technique [15]. All incoming characters are pre-decoded in a centralized decoder, properly shifted to amount for their relative positioning, and subsequently AND-ed to produce the match signal of each pattern. This way, the character comparators are shared through the decoder, while each shifted decoded character value can feed multiple pattern matching modules (AND gates). In addition, the entire module as well as the header matching module are fine-grain pipelined in order to increase its operating frequency. Since the header matching involves comparisons of fixed location fields in each packet, the overall throughput of the packet pre-filtering module is determined by the throughput of the partial pattern matching. Consequently, in order to increase the performance of the packet pre-filtering, the pattern matching module can process multiple incoming bytes per cycle and increase accordingly the overall throughput. In our current experiments (section 5) the pre-filtering mod-



(a) Basic 2-input module

(b) Four Match Example

Figure 2: Priority encoder implementation details and a small example.

ule matches only static patterns; however, it can be extended to support also regular expressions. In case the first payload content of a rule is a regular expression, the pre-filtering module could match a first part (prefix) of the regular expression, which may include wild cards or any other syntax features. The full match engine would then be responsible to match the remaining portion of the regular expression.

Bitmask: Both header and partial pattern matching outputs feed a bitmask which indicates all possible matching rules. Each bit of the mask corresponds to a single rule. For some rules, the pre-filtering module may match only the packet header, if no payload patterns are included (e.g. numerical check of some payload bytes might be performed). In this case, the output of the header matching alone determines the value of this bit in the bitmask. Other rules may match packets of any header and therefore only matching a payload content may determine the outcome of the filter. Furthermore, in case of a rule which needs both the header and the payload pattern, a subsequent AND between the corresponding header and payload pattern matching results produce the outcome of the bitmask. Finally, when the header and pattern matching performed in pre-filtering module is equivalent to a complete IDS rule, this rule should be directly reported and no further matching is required.

Priority Encoder: The bitmask feeds a priority encoder, which outputs sequentially all the positions of the active bits in the bitmask (possibly matching rules IDs). Our priority encoder is fine-grain pipelined and therefore scales well in terms of performance as the number of inputs increases. Figure 2(a) depicts the basic building block of the priority encoder that selects one out of two inputs to be encoded in the output. The above basic block is used to construct the binary-tree-like structure of the priority encoder. In the first pipeline stage, the `valid` and `encode` inputs are the same bit. In each pipeline stage of the encoder, an input is selected over the other to be sent out (when ever possible). When a partially encoded value is forwarded to the next pipeline stage, then is subsequently deleted from the previous stage. To do so, we use extra logic to produce "load enable" signals for the registers of every pipeline stage. Consequently, *all* the inputs of the priority encoder are encoded and forwarded to the output based on their priority/position, and reported sequentially. Figure 2(b) illustrates an example of a 4-input priority encoder. In order to accomplish fine-grain pipeline an encoded value of stage *N* cannot be deleted/overwritten by the next value coming from the stage *N-1* before it is verified that is forwarded in stage *N+1*. Therefore, each input is reported in the output of the priority encoder for 2 cycles. However, in our packet pre-filtering design this is not an issue, since only a few rules are expected to partially match, and this way we achieve performance scalability for large bitmasks.

Our proposed packet pre-filtering shares common ideas with the Piranha approach [1]. However, there are several important differences between the two. First, Piranha is a software-based technique targeting only on IDS payload pattern matching, and does not use full header matching information as we do. Second, our pre-filtering technique works for all search pattern lengths from 1 up to the maximum supported, and provides exact matches for rules up to that length. The Piranha approach is restricted for performance reasons to 4 character (1 word) substrings. The implication of this restriction is that increasing the width in Piranha is likely to reduce efficiency since shorter rule patterns are not covered and at the same time increase processing requirements, while reducing the width would again reduce efficiency due to increased conflicts. Our approach does not suffer from these constraints, but only from an increased implementation cost as the prefix length increases.

4. INTEGRATING PRE-FILTERING IN AN INTRUSION DETECTION SYSTEM

In this section, we describe integrating techniques of the proposed packet pre-filtering module into a complete Intrusion detection Engine. The pre-filtering module will actually match the first part of each IDS rule (i.e. header and partial payload pattern). Consequently, the rest of the system is required to match the remaining parts of the detected possibly matching rules, instead of matching again the parts already matched in the pre-filtering stage.

In software based platforms, the packet pre-filtering module reports the ID of the rules needed to be fully matched, and subsequently, software is employed to continue the processing of these rules. As described below in section 5.1, at least 99% of the rules can be excluded and therefore, the workload of the software can be significantly reduced.

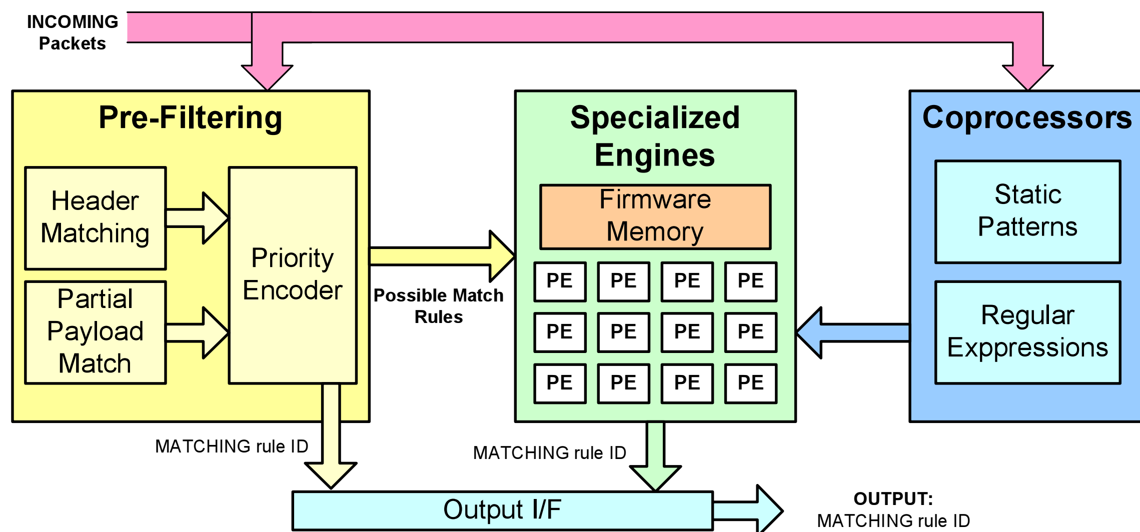


Figure 3: A Reconfigurable Intrusion Detection System utilizing packet pre-filtering.

Figure 3, depicts the block diagram of an envisioned reconfigurable IDS, part of which has been implemented and presented in section 5.2. Incoming packets enter the packet pre-filtering module, which detects the possibly matching rules. In case there are already detected rules in this stage, they are reported. The IDs of the detected candidate rules (that require further processing - full match) are sent in an array of specialized engines. Subsequently, for each one of these rules a firmware is downloaded to a single processing engine, which is reserved to fully match the rule. A processing engine (PE) is reserved whenever a rule is detected by the pre-filtering stage, and released in case of a match, mismatch or end of packet. In case there are no available PEs to match a rule, then the packet should be reported and afterwards the policies set by the system administrator should be applied. We should note here, that this can only occur when a single packet partially matches the descriptions of multiple rules (more than the threshold defined by the system designer i.e. 32 PEs). Each PE keeps track of the stages a rule should pass through in order to produce a match. The processing engines do not perform payload pattern matching. Instead, centralized coprocessors are utilized to match *all* the static patterns and regular expressions included in the IDS ruleset. An interface between the coprocessors and the PEs should be used to feed the PEs with the payload matches and their exact position in the packet payload. Finally, the ID of a rule matched by a PE is reported at the output.

5. EXPERIMENTAL RESULTS

In this section, we present experimental results of our packet pre-filtering approach. First, we utilize the DefCon traces [17] to evaluate the effectiveness of our proposal. We then provide implementation and performance results of the packet pre-filtering module using Xilinx Virtex2-4000-6 and Virtex4-40-12 FPGA devices and preliminary implementation results of a complete intrusion detection engine.

5.1 Simulation Results

To evaluate the effectiveness of our proposed packet pre-filtering module we use trace-driven execution. We use the Snort v2.4 ruleset and Defcon11 traces. For each Snort rule, the pre-filtering module matches the header of the packet (IP header as well as the TCP/UDP headers (source and destination ports), and the ICMP header) and a prefix of a payload pattern (whenever included in a rule). In our experiments, we match 2, 4, 6, 8 or 10 prefix characters of the payload pattern. When the rule payload pattern is shorter than the prefix length, then the entire, exact pattern is matched.

The Snort v2.4 ruleset consists of a total of 3191 rules, out of which 2271 rules (or 71.2%) require content matching, while the remaining 920 rules (or 28.8%) check only header parameters. The rules were grouped into 381 rule sets using a fine-grained header classification that takes into account up to 5 fields as described in section 3. For our tests, we configured the symbolic addresses of the rule-set in the following manner: $HOME_NET was set to the /24 (or a class-C) subnet matching most traffic in the used traces, $EXTERNAL_NET was set as !$HOME_NET, i.e. anything but the home net, and all other symbolic addresses (apart from $AIM_SERVERS) were the same as $HOME_NET.

We evaluated our architecture using Defcon11 traces. Defcon11 contains 9 trace files, 10 million packets in total, out of which 4.6 million packets have payload. The mean payload length is 698 bytes and the maximum payload length is 1460 bytes. Figure 4 plots a break-down of the total number of packets according to the trace files, and also distinguishes between packets with and without payload.

During the IDS execution, each packet activated an average of 3.7 *sets of rules* (or header groups). This result is interesting since it shows that even complete header matching is not refined enough, and leaves opportunities for our pre-filtering step to refine the search space. In addition, out of the 2271 rules that specify pattern matching, the header classification process of the packet determined that an average of ∼45 rules (1.9%) were applicable to be content matched. The maximum number of rules that required string matching for a single packet is an impressive 142 rules (4.5%). Figure 5 shows the pattern length cumulative distri-

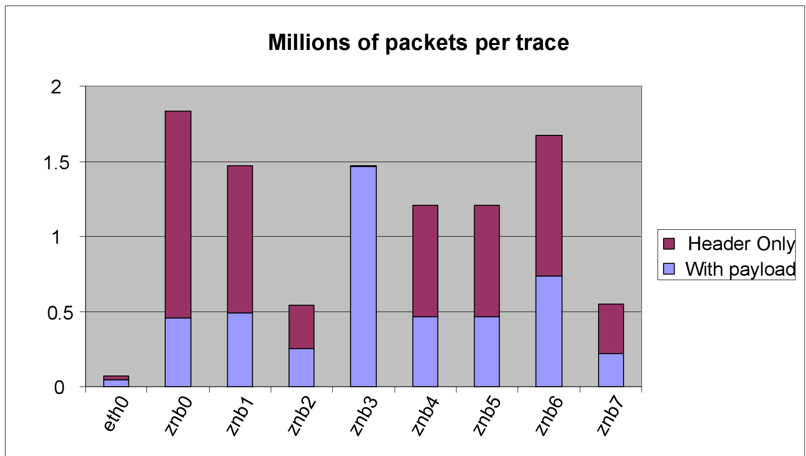


Figure 4: Packet trace statistics: number of packets that include payload and header-only packets in Defcon11 traces.

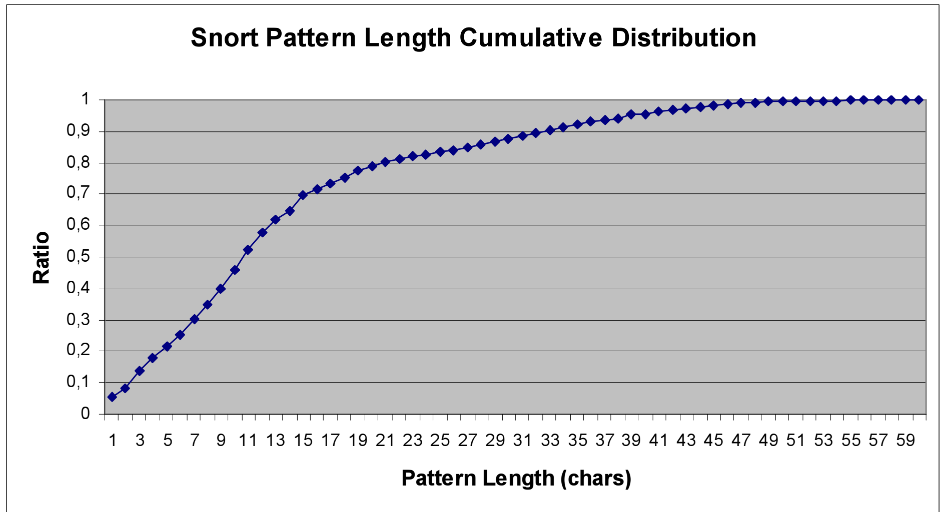


Figure 5: Cumulative Distribution of payload pattern length in the SNORT rules.

bution in our Snort rules. We can observe that more than half of all patterns have length above 11 characters, and that 10% of the rules have length exceeding 34 characters. These numbers show that we would need very wide prefix lengths to guarantee exact matching in the majority of the rules. Small prefixes, in the range of the ones we consider, achieve exact match for 40% of the rules for prefix of size 10, a value that drops quickly to 30% for length of 8, and 20% for length 4 characters. This full matching however is one of the advantages of our prefiltering technique, since full matches can be directly reported and avoid loading the heavier full-match module that would provide no additional useful information.

The following two figures show the effectiveness of the pre-filtering technique. Figure 6 shows the average number of patterns that are determined as eligible or candidate for a full match by our pre-filtering step. As mentioned earlier, we consider pattern prefixes of 2, 4, 6, 8 and 10 characters. The pre-filtering result is a small number of rules which are eligible that depending on the trace and the prefix length ranges from below 1 up to 10 rules. It is noteworthy that, when matching more than 2 payload pattern bytes the average number of candidate rules is significantly reduced down to about 1-3 rules per packet. For 8-character prefix width the maximum value across all traces is only about 3 candidate rules, while the overall average is 1.8 rules. This corresponds directly to the amount of work that the full-match module would have to perform to determine the final match. The effect of pre-filtering is more than an order of magnitude reduction in the number of candidate rules compared to simply using header match information which would have checked 45 rules.

Figure 7 shows the maximum number of patterns that were indicated as candidates by our pre-filtering according to the trace files and the prefix length. This results are important to measure the maximum amount of work that the full match module will experience. For a best-effort implementation that can delay packets while the processing is not completed, these maximum values offer an indication of the maximum jitter in packet latency. For a fixed latency

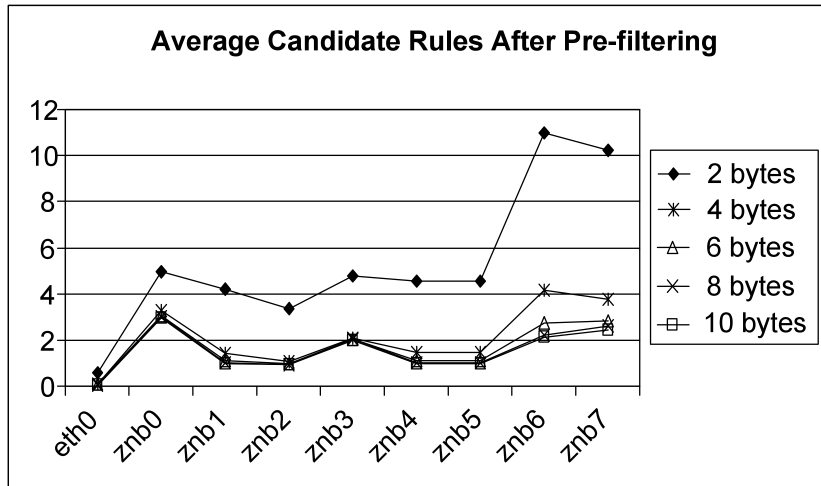


Figure 6: Average number of candidate rules per packet after the pre-filtering step as a function of the pre-filtering length.

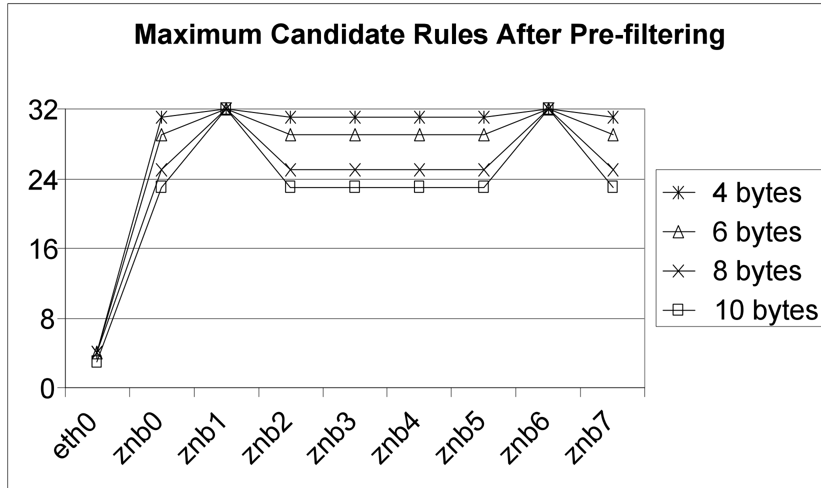


Figure 7: Maximum number of candidate rules per single incoming packet after the pre-filtering step as a function of the pre-filtering length (length 2 was omitted for clarity due to exceedingly large values).

implementation, the maximum values indicate the degree of parallelism that must be provided to guarantee maximum processing throughput under all circumstances. We note that for small prefixes the maximum values are indeed very high (up to 63 when prefix length is 2 bytes). However for more than 2 character prefixes, all trace files result in a maximum value of no more than 32 rules. We should emphasize again here that these numbers are rare since the average numbers are much smaller, and indicate the worst and infrequent case.

The above analysis indicates that (i) pre-filtering can be very effective in reducing the search space for a full-match module, and (ii) that it can achieve this goal using relatively small prefix lengths. In general, larger prefix length would result in marginally better results but at increased pre-filtering cost. Narrower prefixes will reduce the pre-filtering cost, but increase the load of the full match unit. Since the pre-filtering performance does not improve noticeably when increasing the prefix length from 8 to 10 characters, we believe that a good tradeoff for the prefix length is 4-8 characters and we proceed to evaluate the system implementation cost and performance with a prefix length of 8 characters.

5.2 Implementation Results

Next, we present the implementation results of two packet pre-filtering designs. In addition, we estimate the area requirements of a complete intrusion detection detection engine as described in section 4.

For the evaluation of our packet pre-filtering designs we used the Xilinx ISE 8.1 tools for synthesis and place and route operations. We used two devices from different FPGA families: a Virtex2-4000-6, a Virtex4-40-12. The internal cell structure of these devices is similar and the results both for the area cost are practically identical. Table 2 lists the area cost of our designs in terms of flip-flops, LUTs and total device slices. The header matching part of the design involves comparisons between numerical values and fixed lo-

Table 3: Payload Scanner Coprocessors: matching Regular Expressions and Static patterns.

Description	Input bits /cycle	Device	Throu-ghput (Gbps)	Slices	LC/char	MEM Kbits	# RegExp *or* Patterns
RegExp Engine [3] Regular Expressions	8	Virtex2 -4000/ Virtex4 -40	2/2.9	12,537	1.39	0	509
Static Pattern Matching [16]				4,733	0.28	630	2,188
***Coprocessors* (Total)**	8	Virtex2/ Virtex4	2/2.9	17,270		630	509+2,188

Table 2: Packet Pre-filtering Area Cost.

Module	FFs	LUTs	Slices
Header Field Extractor	120	49	64
Header Matching	1352	778	946
Static Pattern Matching DCAM [15] (8 bits/c.c.)	3,226	2,929	1,688
Static Pattern Matching DCAM [15] (32 bits/c.c.)	12,164	11,276	6,103
Priority encoder	12,804	15,986	8,020
Control	112	112	56
Total (8 bits/c.c.)	**17,614**	**19,854**	**10,774**
Total (32 bits/c.c.)	**26,552**	**28,201**	**15,189**

cation fields of the packet which is implemented in parallel. Consequently, the supported throughput of our designs is determined by the payload pattern matching module. We implemented two alternative pattern matching designs to be integrated with the rest of the packet pre-filtering module. The first one processes one incoming payload byte per cycle, while the second processes four bytes per cycle and thus supports higher throughput (about 4× higher). The overall area cost of the packet pre-filtering module is 10,774 and 15,189 slices for 8 and 32-bits datapaths respectively, which easily fits in a small/medium FPGA device[1]. Table 2 also offers a break-down of this area cost per module, and as expected the majority of the flip-flops and logic is consumed by the payload pattern matching module and the priority encoder. The total cost is dominated by the priority encoder, since a 3,191-bit bitmask (which corresponds to 3,191 Snort v2.4 rules) has to be encoded. The priority encoder is about 75% of the entire pre-filtering module, when 8-bits per cycle are processed, and 50% when the datapath width is 32 bits. All the packet pre-filtering sub-modules are fine-grain pipelined and therefore the operating frequency of the designs is relatively high: 335 MHz (8-bits/cycle) and 303 MHz (32-bits/cycle) for Virtex2-4000-6, supporting 2.7 and 9.7 Gbps throughput respectively. For Virtex4-40 the performance is about 50% higher, that is 4 and 14 Gbps for 8 and 32 bits datapaths respectively.

Apart from the packet pre-filtering, an intrusion detection engine, as depicted in figure 3, consists of the payload matching coprocessors and the specialized engines. In our previous work, we have designed and implemented the regular expressions and static pattern matching modules [3, 16]. Table 3 shows the post place & route performance and area results of the coprocessor modules which match the payload regular expressions and static patterns included in the ruleset of Snort v2.4. When processing 8 bits per cycle the coprocessor is able to support 2 Gbps throughput and requires about 17,000 slices. For 32-bit datapaths, our preliminary results show that the area cost of the coprocessors would be about 65K slices and 1,4 Mbits memory. Finally, valuating the cost of the specialized engines, we can estimate that a complete IDS design which processes 8-bits per cycle would require about 35K slices and 3-5 Mbits Block RAM, while for 32-bit datapaths it would occupy about 90K slices and 4-6 Mbits RAM. The above designs would be able to fit in a single current FPGA device.

[1]Current Xilinx (Virtex4) FPGA devices contain up to 90,000 Slices.

6. CONCLUSIONS

We have presented *Prefiltering*, a powerful hardware-based technique aim to reduce the processing requirements for intrusion detection. We claim that implementing the header matching portion of a NIDS system together with a small prefix match (in the range of 4-8 characters) can eliminate most of the rules and determine a handful of applicable rules, that can then be checked more efficiently by a full-match module. The technique is amenable to various kinds of parallelism at the full-match module whether implemented in hardware or in software.

We also implement the pre-filtering and the full-match modules, including support for the more recent Snort features such as regular expressions. We analyze the cost of all components and show that the entire system would fit in a current high-end FPGA device, while achieving processing bandwidths of about 2.5 and 10 Gbps (Virtex2) and 4 and 14 Gbps (Virtex4) processing one character and four characters per cycle respectively.

We can further optimize pre-filtering by supporting variable prefix length per rule. This flexibility would allow us to select an "optimal" prefix length at the rule granularity with two potential benefits: (i) cost savings from smaller lengths when possible, and (ii) better accuracy when selectively longer prefixes are used. Furthermore, we can generalize the Piranha approach [1] and select arbitrary substrings instead of prefixes as we do in this paper, possibly improving the accuracy of pre-filtering. However, this improvement will come at a cost, since in this case the full match unit will have to re-scan the packet for scratch, while in our prefix implementation it can pick up from the last matched character. Finally, we plan to evaluate the impact of the more advanced rule structures (distances, regular expressions, etc) on the complexity and performance of intrusion detection systems.

Acknowledgements

The authors thank Evangelos P. Markatos and Michalis Polychronakis for their help on Defcon traces.

7. REFERENCES

[1] S. Antonatos, M. Polychronakis, P. Akritidis, K. D. Anagnostakis, and E. P. Markatos. Piranha: Fast and memory-efficient pattern matching for intrusion detection. In *Proceedings 20th IFIP International Information Security Conference (SEC 2005)*, May 2005.

[2] Z. K. Baker and V. K. Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection systems on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[3] J. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis. Regular Expression Matching for Reconfigurable Packet Inspection. In *IEEE International Conference on Field Programmable Technology (FPT)*, 2006.

[4] C. R. Clark and D. E. Schimmel. Scalable Parallel Pattern-Matching on High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[5] V. Dimopoulos, G. Papadopoulos, and D. Pnevmatikatos. On the importance of header classification in hw/sw network intrusion detection systems. In *Proceedings of the 10th Panhellenic Conference on Informatics (PCI)*, November 11-13, 2005.

[6] M. Fisk and G. Varghese. An Analysis of Fast String Matching Applied to Content-based Forwarding and Intrusion Detection. In *Techical Report CS2001-0670 (updated version)*, University of California - San Diego, 2002.

[7] R. Franklin, D. Carver, and B. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[8] E. Markatos, S. Antonatos, M. Polyhronakis, and K. G.Anagnostakis. Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN)*, pages 146–152, November 2002.

[9] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[10] G. Papadopoulos and D. Pnevmatikatos. Hashing + Memory = Low Cost, Exact Pattern Matching. In *Proceedings of 15th International Conference on Field Programmable Logic and Applications*, 2005.

[11] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[12] SNORT official web site. http://www.snort.org.

[13] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using fpga. In *FPGA*, pages 238–245, 2005.

[14] Sourcefire. Snort rule optimizer. In *www.sourcefire.com/whitepapers/sf_snort20_ruleop.pdf*, June 2002.

[15] 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*, April 2004.

[16] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. A Reconfigurable Perfect-Hashing Scheme for Packet Inspection. In *Proceedings of 15th Int. Conf. on Field Programmable Logic and Applications*, 2005.

[17] The Shmoo Group: the Capture the Flag Data. http://cctf.shmoo.com/.