A Framework for Rule Processing in Reconfigurable Network Systems

Michael Attig and John Lockwood [*]
Department of Computer Science and Engineering
Washington University
Saint Louis, MO 63130
E-mail: {mea1, lockwood}@arl.wustl.edu

Abstract

High-performance rule processing systems are needed by network administrators in order to protect Internet systems from attack. Researchers have been working to implement components of intrusion detection systems (IDS), such as the highly popular Snort system, in reconfigurable hardware. While considerable progress has been made in the areas of string matching and header processing, complete systems have not yet been demonstrated that effectively combine all of the functionality necessary to perform rule processing for network systems.

In this paper, a framework for implementing a rule processing system in reconfigurable hardware is presented. The framework integrates the functionality to scan data flows for regular expressions, fixed strings, and header values. It also allows modules to be added to perform extended functionality to support all features found in Snort rules. Reconfigurability and flexibility are key components of the framework that enable it to adapt to protect Internet systems from threats including malicious worms, computer viruses, and network intruders.

To prove the framework viable, a system has been built that scans all bytes of Transmission Control Protocol/Internet Protocol (TCP/IP) traffic entering and leaving a network's gateway at multi-gigabit rates. Using Xilinx FPGA hardware on the Field programmable Port eXtender (FPX) platform, the framework can process 32,768 complex rules at data rates of 2.5 Gbps. Systems to handle data at 10 Gbps rates can be built today using the same framework in the latest reconfigurable hardware devices such as the Virtex 4.

1. Introduction

High performance network intrusion detection and prevention systems are desperately needed in order to protect the Internet from malicious attacks [1, 2]. The use of software alone to perform intrusion prevention has been shown to be too slow for use in networks, even on relatively slow, 100 Mbps links [3, 4]. Reconfigurable hardware has been shown effective at performing many of the computationally intensive tasks needed for intrusion detection and prevention. Research groups have designed high performance systems that perform regular expression scanning, static signature scanning, header processing, and TCP flow reconstruction. However, with few exceptions [5, 6], the systems implemented only performed single aspects of the task. The network security community needs all of the aspects of rule processing to be integrated in order to deliver a viable solution for useful protection.

Complex rule sets, such as those found in intrusion detection systems like Snort [7], process rules that specify the processing of packet headers, the matching of patterns in packet payloads, and the action to take when a rule is matched. A Snort rule, in fact, has a syntax like:

> alert tcp any 110 → any any (msg:“Virus - Possible MyRomeo Worm”; flow:established; content:“I Love You”; classtype:misc-activity; sid:726; rev:6;)

The rule above indicates that the packet header can match a wildcard value for the source IP address, the destination IP address, and the destination port. However, the source port of the packet must have the value of 110. The rule also specifies that the protocol should be TCP. The second part of the rule specifies to search for *“I Love You”* over an established TCP/IP connection. If the signature were found in an UDP packet, it is not a match. Flow reconstruction across multiple TCP packets must be performed, as there is no guarantee that the string is not segmented across multiple packets. All of the tasks described above must be performed just to

[*] This work was supported by a grant from Global Velocity. The authors of this paper have received equity from the license of technology to that company. John Lockwood has served as a consultant and co-founder to Global Velocity. `http://www.globalvelocity.com`

process this single rule. There are 2464 rules in version 2.2 of the Snort rule database from September 2004. Over 80% of the rules require performing all steps like those in the rule above. The remaining rules require some combination of the above steps.

Once the tasks of rule-processing have been defined, they need to be integrated and implemented. In this paper, a framework for performing rule processing in hardware is proposed that utilizes modular interfaces to enable plug-and-play integration of the multiple components of rule processing. A complete implementation of Snort features can be implemented in this framework to perform rule-based processing at multi-gigabit speeds.

Some features of Snort have not been implemented in reconfigurable hardware yet, such as support for pattern matching at specified offsets. This framework allows integration of modules so experimentation with future processing modules can be performed. Additionally, the best-known methods for header-processing and content-scanning developed to date can be used concurrently. A module that efficiently matches on multiple headers can be used in tandem with one that is optimized for specific headers. Similarly, a module that is optimized at detecting regular expressions can process data directly before a module that is optimized for static signatures. Through the use of modules, additional features can be added as they are developed.

2. Related Work

State-based intrusion detection and prevention systems (IDPS) require three primary elements: TCP flow reassembly, header processing, and content scanning. Using information received about these elements, a rule processor determines whether rules match.

2.1 TCP Flow Processing

Schuehler and Lockwood developed a TCP Processor in FPGA logic that annotates control information onto incoming IP packets, specifying where headers begin and end and where payload data begins and ends [8, 9]. The TCP Processor is capable of simultaneously keeping state for 8 million TCP flows while operating at 2.9 Gbps in the FPX platform. TCP data is presented to processing modules in-order along with pertinent flow information.

Nguyen, Zambreno, and Memik created flow monitor units (FMU) that provide lower level components with flow-based information [10]. Their results show the ability to process a flow at very high throughput.

Necker, Contis, and Schimmel implemented a single TCP-stream assembler in FPGA technology capable of operating at 3.2 Gbps [11].

The need for TCP offload engines (TOEs) has become greater as network link speeds increase [12]. A TOE processes the layers of TCP/IP stack such that a host need not perform protocol related computations.

2.2 Header Processing

Header processing and packet classification have been extensively studied in the literature. Header processing in the context of intrusion detection requires that all matching header rules be investigated. A simplification provided by operating on TCP flows is that header processing only has to be done once per flow.

Yu and Katz used ternary content addressable memories (TCAMs) to return multiple matching packet headers [13].

Song and Lockwood used a hybrid bit-vector and TCAM algorithm to compress the matching header representation [14]. The authors converted 222 Snort header rules into 264 trie-node prefixes and 33 distinct TCAM entries.

2.3 Content Processing

There are many techniques to perform string matching that are well suited for FPGAs [15]. Focusing on efficiency, resource consumption, and module throughput, several groups have invested a significant amount of effort to improve string matching. Any of the following circuits could be used as content processing modules in this framework.

Baker and Prasanna developed a technique to partition signature databases into independent pipelines, allowing FPGA resources to be efficiently utilized by reducing redundancy [16].

Clark and Schimmel reduced the redundancy inherent in string matching when using non-deterministic finite automata (NFA) and increased the throughput of string matching by processing multiple characters per clock cycle [17].

Moscola et. al created an automated way of generating deterministic finite automata (DFA) structures optimized with JLex in order to process regular expressions [18]. It was found that, in most cases, the number of states necessary to implement the DFA was comparable or less than the number needed for an equivalent NFA.

Cho, Navab, and Mangione-Smith created a content-based firewall using discrete logic filters [19]. They created automated techniques to generate highly parallel comparator structures that can be quickly configured. This work was expanded to include logic re-use and read only memory.

Sourdis and Pnevmatikatos created unique VHDL instances for each signature to process [20]. Signatures are added or removed by modifying the instance loaded. They achieved high-throughput using deeply pipelined comparators and encoders, as well as by reducing fan-out.

Sugawara, Inaba, and Hiraki implemented a string matching method using trie-based hashing in order to achieve high throughput [21].

Bloom filters, using several hashes, have been shown to be able to store thousands of search strings [22]. The logic footprint is constant, regardless of the search criteria. Updates are performed by dynamically re-writing values in FPGA block RAMs.

Gokhale et. al investigated a hardware/software approach to intrusion detection, where header and content vectors of matches are sent to software [6]. This system separated string matching and rule processing onto separate environments, the former in hardware and the latter in software, and was capable of supporting a few hundred rules.

2.4 Our Solution

Rule processing is not solely composed of header processing nor solely of content scanning. It is the combination of these two aspects that defines rule processing. We have developed what we believe to be one of the first techniques to combine the above elements in FPGA hardware to support thousands of Snort-like rules. Previous techniques have sent match vectors of the two components to software to perform the rule processing [6]. Our solution involves sending header and content match IDs to a rule processing circuit. The rule processor, operating in real-time, determines whether a rule matched.

3 Rule Processing Framework

3.1 Elements of Rule Processing

Data flows through the system from a TCP flow assembler to *h* header processors and *c* content scanners, as shown in Figure 1. The flow assembler provides the in-sequence ordering of TCP data required by content scanning engines. Header processors perform packet classification, determining which header rules match the incoming packet. Content processors scan payload bytes of data streams for regular expressions and static strings. Strings can appear anywhere within the payload or even across packet boundaries. After finding matching headers or signatures, the header and content modules forward match IDs to the rule processor.

A key aspect of this IDPS is flexibility. The nature of future attacks to the Internet's infrastructure are difficult to predict. Network security devices that make the Internet safe require constant change. As a result, the use of reconfigurable hardware is highly desirable. The reconfigurability of hardware allows the system to adapt to new threats.

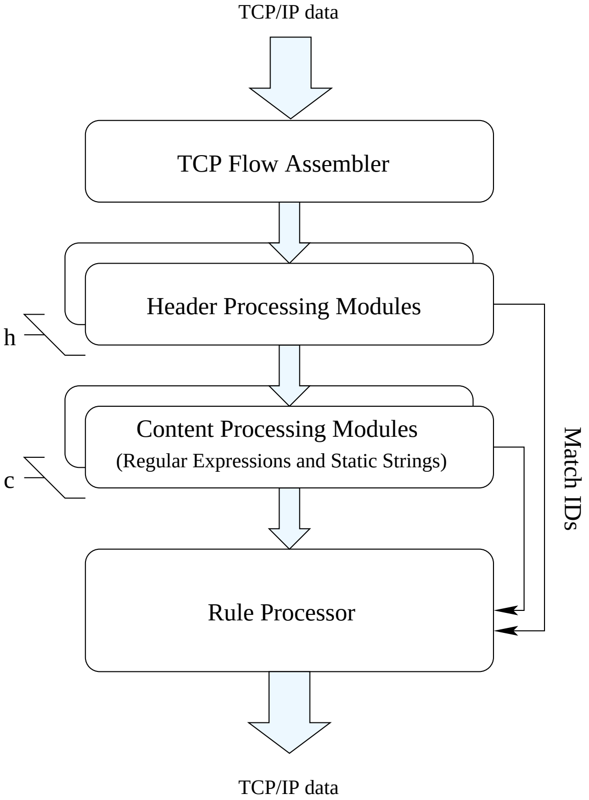

TCP/IP data
TCP Flow Assembler
Header Processing Modules
h
Content Processing Modules
(Regular Expressions and Static Strings)
c
Match IDs
Rule Processor
TCP/IP data

Figure 1. The rule-processing framework allows header processing and content scanning components to be added or removed without affecting rule processing. A standardized communication mechanism was adopted to integrate multiple components.

3.2 Interface

To compose a modular system, a standardized communication format must be used between data processing modules. Data processing modules may be on separate devices or in different regions of a FPGA. Modules should have the property that they can be added, removed, or reconfigured without interrupting the rest of the rule processing system. A standard interface was developed to transit data between modules via a communication wrapper. The communication wrapper, as shown in Figure 2, transports data to and from data processing modules.

The interface signals include: start of data (*sod*) to indicate when a new data stream enters the system; a data enable signal (*en*) to indicate when data is present; an end of data (*eod*) signal to indicate when there is no more data; the *data* of the flow itself; the number of valid bytes (*vb*) on the *data* signal; and a busy signal (*stop*) used to temporarily halt the flow of data in the case of a backlog.

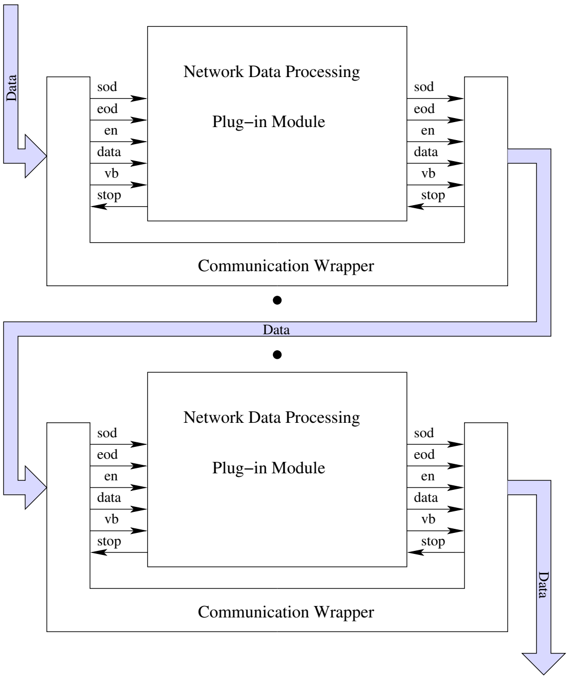
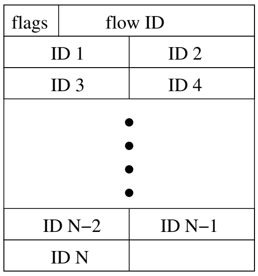


Figure 2. The communication wrapper provides a mechanism to communicate information between modules and the rule processor. The wrapper abstracts away the underlying transmission requirements needed to communicate between modules across devices or across the FPGA.

A module can be added to the system by adhering to the following two properties. First, the module must be able to accept and act upon IP packet data. Second, the module must be able to provide relevant match IDs (either header or content) to the rule processor using the communication wrapper. Each unique header rule and signature are given an unique identification number. When a header or signature matches, the module informs the rule processor of what IDs matched using the data format of Figure 3.

A standard data format was defined to communicate information about a flow to the rule processor. The flow ID informs the rule processor of which TCP flow the results belong. Note that in the case of stateless protocols, such as UDP, this flow ID is given the default value of 0. The remaining words contain a list of what particular strings, regular expressions, or header rules matched. Several IDs could potentially match in a packet. The module designer can choose whether to buffer matches found and burst them out at the end of a packet or to send match information as it becomes available.



flags	flow ID
ID 1	ID 2
ID 3	ID 4
• • • •	
ID N−2	ID N−1
ID N	

Figure 3. The data format for communicating matching rules and signatures consists of flags, a flow ID field, and a list of matching ID numbers.

Using this interface, a module designer can implement any feature found in Snort and interface with the rule processor by informing it as to which header or content matched by providing the ID of the matching criteria. For example, many content rules are only supposed to match if they are found within a certain range of the payload. This is difficult for a generic pattern match circuit to implement. However, a module could be developed to handle this case. Instead of loading the generic pattern match circuit with these signatures, the new module could be loaded with these signatures. Then, when the signature is found in the specified range, the unique content ID given to the signature is forwarded to the rule processor by the new module.

4 Rule Processor Design

There are aspects of rule processing that are critical for networking systems. First, the system must allow new rules to be added as quickly as possible. Second, the system must be able to support large numbers of rules. There are 2464 rules in Snort version 2.2, but the number of rules is expected to increase. Third, the system must be able to operate on TCP flows. This point implies there must be an efficient way to retain context information for the flow. Finally, the system must operate in real-time. A response to an intrusion must be detected and blocked at the instant that it occurs.

4.1 Analysis

Analysis of systems should consider the number of unique headers and signatures that the systems can handle, the number of different header rules that can be processed, and the number of signatures that can be associated with a given rule.

Rule database analysis should consider how often signatures occur and how many rules solely consist of a header rule. Examining this helps to determine an optimal amount of resources for a practical system.

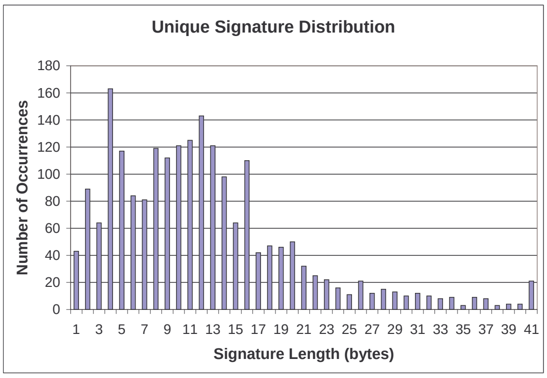


Figure 4. The number of signatures associated with each length. Note that signatures longer than 40 bytes have been lumped into a single bar.

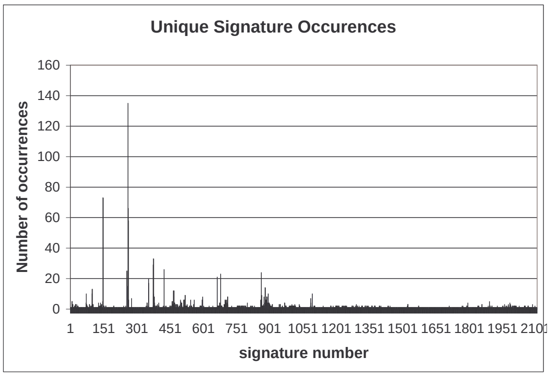


Figure 5. The distribution of how many times a signature appears in an unique Snort rule. Only 18 signatures appear in more than 10 rules.

The Snort rules database contains 292 unique header rules, 2107 unique static signatures, and 233 regular expressions. Most Snort rules give a header rule of the form *EXTERNAL_NET* to *INTERNAL_NET*. Up to 10 of the 292 headers can simultaneously match any given packet. The static signatures are distributed across the range 1 to 122 characters, as shown in Figure 4. The bulk of the distribution is below 41 bytes. These 2107 signatures are spread across 2296 of the 2464 rules. Figure 5 shows how often signatures appear in multiple unique Snort rules. The y-axis is how many times the signature is found in a rule, and the x-axis gives a number to each of the 2107 signatures. Only 18 signatures occur in more than 10 rules. The hexadecimal signatures |*00 00 00 00*|, |*01*|, and |*00 01 86 A0*| occur in 135, 73, and 66 different Snort rules, respectively.

4.2 Functionality

The rule processor collects information on the matching headers and signatures from the modules. Using the communication wrapper interface, modules can specify what has been found in each flow as data passes through it.

The rule processor contains logic to allow rules to be added, deleted, or modified. To the rule processor, the syntax of the rules takes the form:

$$< action > \; H_{ID} \; \wedge \; (C1_{ID} \; \wedge \; C2_{ID} \; \wedge \; \cdots \; \wedge \; Cn_{ID})$$

Thus, a rule consists of an action, a header rule, and 0 to *n* signatures. A rule matches when the header rule matches and each of the signatures specified, if any, are detected. When a match is found, the action specified is taken. Rules are programmed into the system dynamically using control packets from a management console or the network.

4.3 Architecture

With these observations considered, the rule processing architecture of Figure 6 was developed to perform rule processing as traffic flows through a reconfigurable hardware circuit. To achieve high performance, the circuit is pipelined with seven stages. All communication is performed via the communication wrappers. The rule processor handles two forms of inputs and a single form of output. Programming information from a control host enters on the control interface. Matching header and content IDs enter on the ID interface. Rule match information and actions to take are output the alert interface.

4.3.1 Stage 1: Input

The input stage has two main components, one for each of the two types of inputs. The control FSM receives programming information from software for adding, deleting, or modifying rules. Software can read/write SRAM, read/write on-chip block RAM, or query system event counters. On the other interface, matching header and content IDs enter, where they are buffered in a FIFO. From this FIFO, they enter the processing pipeline. This FIFO acts as the flow control. Information is held here in the case of performing context switching. Header IDs (HIDs) and content IDs (CIDs) fork apart in this stage. CIDs advance to stage 2 of the pipeline, while HIDs are forwarded to stage 5.

4.3.2 Stage 2: Content ID Check

Matching CIDs enter into this stage to determine if this content has already been seen for this particular flow. A large bit-vector, stored in on-chip block RAM, is directly

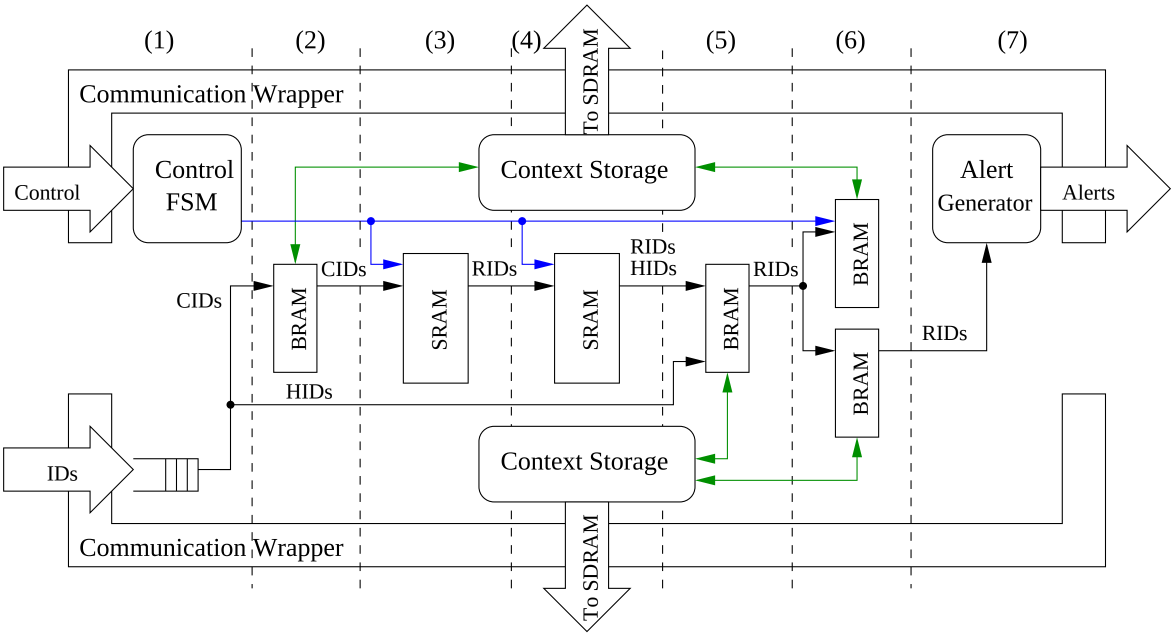


Figure 6. The rule processor consists of seven pipeline stages: (1) input, (2) content ID check, (3) rule ID retrieval, (4) header ID mapping, (5) header ID check, (6) count check, and (7) alert output.

indexed by the CID value. If the indexed location is set, no further processing is performed because the signature was already detected, but a rule has not matched yet. If the location is not set, the CID is passed to stage 3 and the bit is set. Note that upon a rule match, stage 6 clears the signatures found in the matching rule from this index so the rule can be allowed to match again. Only the first occurrence of a CID in a flow results in the CID being forwarded to stage 3.

4.3.3 Stage 3: Rule ID Retrieval

Stage 3 receives uniquely occurring matching signatures, and a reverse index look-up [23] that maps CIDs to rule IDs (RIDs) is performed. Linked lists are maintained in SRAM, where the first node of a list is directly indexed by the CID. Iterating through the list, each RID associated with the CID is returned. For example, the signature *"I Love You"* is found in a single rule, so only one RID is returned. If *".mp3"* is found, 4 RIDs are returned since *".mp3"* is found in 4 rules. Assuming an uniform distribution of signatures that match, 1.58 RIDs will be retrieved per unique signature match on average.

Memory management is controlled by software in order to simplify the hardware implementation. An entry in SRAM consists of a rule ID and next pointer. The RID is passed to stage 4 while the next pointer, if valid, is examined. The first RID will always be associated with the direct-index of the CID. After that, software maintains a free-list of SRAM words that can be used for next pointers.

4.3.4 Stage 4: Header ID Mapping

Potentially matching RIDs enter stage 4, where the HID associated with the RID is looked up. Using another bank of SRAM, a mapping from RID to HID is maintained. The RID directly indexes into SRAM, and the HID associated with that RID is returned. At this point a similar technique, as used in stage 3, could be adopted if rules were ever to consist of more than a single header rule. However, the current Snort rules do not associated more than one header rule with a rule, so the architecture was simplified. The RID and HID are sent to stage 5.

4.3.5 Stage 5: Header ID Check

This stage is similar to stage 2, only a HID queries the bit-vector instead of a CID. The HID from stage 4 directly indexes the block RAM, checking to see if the HID was found to match this flow. The value in the index is set by forwarding incoming matching HIDs from stage 1. If the header matches, the RID is passed along to stage 6. If the header does not match, the RID is dropped from the pipeline.

The astute reader may have noticed that header-only rules will be neglected in this scheme. To account for these

168 rules, a special ID range was allocated in the bit-vector to inform this stage that when these headers arrive, a header-only rule matched. In these cases, a rule match is declared without any content being processed.

4.3.6 Stage 6: Count Check

When a potential matching RID is passed into this stage, an on-chip block RAM is queried to determine if the number of signatures associated with the rule have been detected. The RID is used to directly index two block RAMs. From the first block RAM comes the count required for a rule to match. This value is programmable by software control packets. From the second block comes the current count. This count is incremented and compared to the requirement for a rule match. If the two are equivalent, the rule with the given RID is declared a match, and the RID is passed to the next stage. When a rule matches, this stage clears the CIDs embedded in this rule from the bit-vector of stage 2 so that subsequent rule matches can be reported as well. The count value in the block RAM is also reset. Enough space is allocated to support rules that contain 15 signatures. Current Snort rules have no more than 7 signatures specified.

Note that a RID will only enter this stage if (1) the first occurrence of one of the signatures found in the rule is detected, and (2) the header specified in the rule matched. This ensures the count will only be incremented once per signature in a rule.

4.3.7 Stage 7: Alert Output

In the final stage matching rules are reported to a control host, where the match can be logged and acted upon. The RIDs that match in a packet are bundled into a single control message that is passed through the communication wrapper to the control host.

4.4 Efficient Context Switching

To facilitate efficient context-switching of the processing circuit on each packet of a different flow, a mechanism to save only pertinent information is required. Stages 2, 5, and 6 perform context switching upon the arrival of different flows. With support for up to 32,768 rules, a full loading of each bit vector would require swapping in and out 192 Kbits of information per flow.

Instead, the amount of storage required is reduced by only storing the CIDs and HIDs of matching criteria for the flow. These are the addresses of set bits in the respective bit-vectors. When the need to perform a context switch arises, buffers of matching criteria are sent to off-chip memory, and the locations specified by the IDs are cleared in the bit-vectors. Thus, when retrieving context information, the returned addresses are used to set the appropriate bits in the bit-vectors. To simplify the design, 512 bytes of storage space are allocated per flow, which can be stored or retrieved in 32 clock cycles.

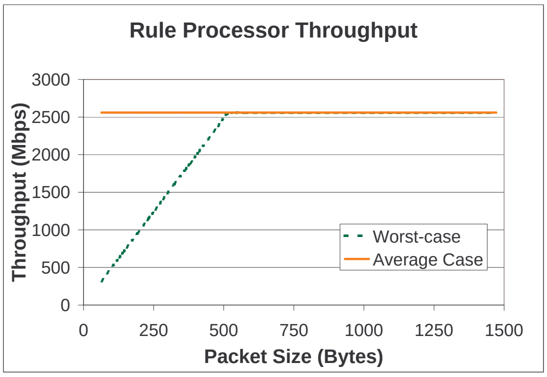


Figure 7. Average-case and worst-case throughput of the rule processor versus the input packet size.

4.5 Evaluation

The system can provide high throughput over all packet sizes in the average case. The lowest system throughput occurs when minimum sized TCP packets arrive belonging to different flows that contain the signature that is present in the most number of rules. For the current rule set, this would occur for TCP packets with 4 bytes of payload containing |*00 00 00 00*|. This results in 135 RIDs being processed.

The average-case and worst-case throughput of the rule processor is plotted in Figure 7 versus the number of payload bytes. Note that as packet size increases for the worst-case signature, the throughput of the system increases to the maximum value since it takes longer to receive the packet.

Since only 18 signatures appear in more than 10 rules, this worst-case performance can be alleviated by using specialized modules to process them. The frequently occurring CIDs are forwarded to these modules instead of into the pipeline. The modules check each HID with which the CID can be associated and inserts only those RIDs into the pipeline that contain matching HIDs. Since at most 10 HIDs can match for a given packet, at most 10 RIDs will be inserted into the pipeline.

4.6 Implementation Results

The rule processor has been implemented using a Xilinx Virtex 2000E FPGA on the FPX platform. The FPX gives access to 2 banks of zero bus turnaround (ZBT) SRAM and 2 banks of 512 MB SDRAM. The FPGA and memory on

Group and Component	Device	Logic Cell %	Throughput (Gbps)
GaTech Stream Assembler[11]	Virtex 1000	10%	3.2[1]
Northwestern U. Flow Monitor [10]	Virtex2-8000	-	48.3[2]
WashU TCP Processor [8]	Virtex4 140	35%	10.3
WashU BV-TCAM [14]	Virtex4 100	10%	10
Crete Pre-decoded CAMs[20]	Virtex2-6000	95%	9.7
GaTech Decoder Trees [17]	Virtex2-8000	81%	7
Tokyo Trie-based Hash [21]	Virtex2-6000	7 %	10
UCLA Packet Filters [19]	Spartan 3 2000	40%	3.2
USC Partitioning [16]	Virtex2 Pro 100	15%	10.3
WashU Bloom Filters	Virtex4 100	85%	20.4
WashU Rule Processor	Virtex4 100	95%	15.9

Table 1. The table above gives the device utilization and throughput for TCP processing, header processing, and string matching components involved in rule processing. Logic cell percentage is based on the number of slices used in a particular device.

LUTs	4,738 (12%)
Slices	4,838 (25%)
Block RAMs	142 (88%)
Input Size	32 bits
Frequency	80.07 MHz
Max Throughput	2.56 Gbps

Table 2. A summary of the resource utilization of the rule processor on the FPX Platform.

this platform were more than adequate to implement the system. Table 2 summarizes the results obtained for the rule processor.

FPGA features were extensively used to improve the efficiency of the rule processor. As Table 2 shows, 142 of the fast, on-chip block RAMs were used and proved to be the critical resource of the circuit. By utilizing these fast memories to hold the large bit-vectors, the throughput of the system dramatically increases while the latency decreases.

4.7 Component Resource Requirements

Rule processing is very resource-intensive task. The constituent components can require an entire FPGA in themselves. Table 1 shows representative requirements for components developed by various groups. The first group shows TCP flow processing components. The next group shows a header processing technique. The largest group shows string matching techniques. The final group shows the rule processor.

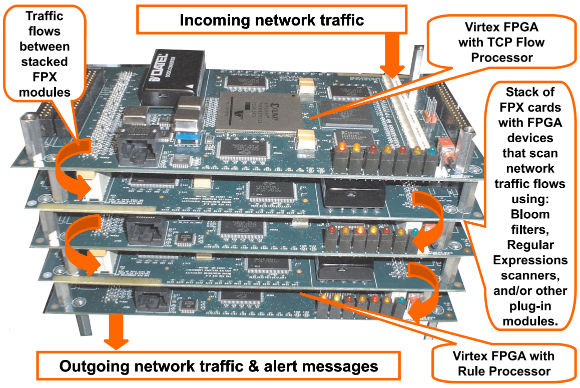

Traffic flows between stacked FPX modules
Incoming network traffic
Virtex FPGA with TCP Flow Processor
Stack of FPX cards with FPGA devices that scan network traffic flows using: Bloom filters, Regular Expressions scanners, and/or other plug-in modules.
Outgoing network traffic & alert messages
Virtex FPGA with Rule Processor

Figure 8. The FPX platform in a stacked configuration allows network processing to occur across multiple FPGAs to perform intrusion detection operations. IP data enters at the top of the system and exits at the bottom. Control information and alerts are sent through the interface on the bottom.

4.8 Implementation

To demonstrate use of this framework, a system was implemented using multiple FPX devices [24]. The framework allows each of the modules to be dynamically reconfigured into the system to perform rule processing functions in a modular way. The system integrates modules that scan for regular expressions and static signatures. These components are configured in the FPX platform in a stacked configuration. Figure 8 shows how the components are arranged in the system.

Internet traffic enters the system via a Gigabit Ethernet line card. Data is processed by the TCP Processor, Bloom

[1] This assumes a single TCP stream.
[2] This assumes 40 Byte packets.

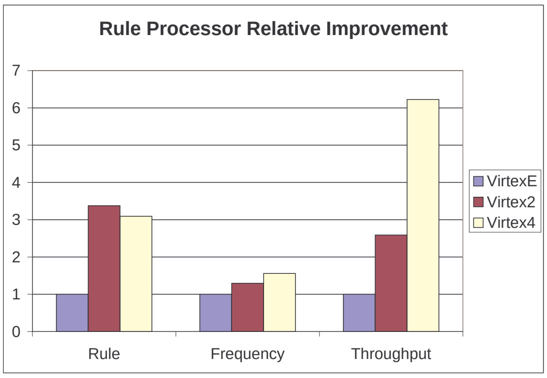


Figure 9. The relative improvements of the rule processor design using newer FPGAs.

filter module, regular expression module, and rule processor, and the data is passed out of the system via another Gigabit Ethernet line card. Matching signatures are detected and forwarded to the rule processor. The rule processor was configured to toggle including header information in the rule match determination in order to test functionality without actually having an implementation of a header processing module.

To make the system more cost effective for wide-scale deployment, the same logic currently implemented in the multiple, stacked Virtex 2000E chips could be integrated together into a single bigger FPGA. Given the size of each circuit and the speed at which it operates on the Virtex 2000E, the entire resulting Snort processing system could fit into a single Virtex 4 FX100. This chip has 94K logic cells (LCs), as compared to the Virtex 2000E's 43K LCs. Additionally, 8 Mbits of block RAM are available, as compared to the Virtex 2000E's 655 Kbits.

To expand the rule processor to operate at higher throughput and support additional rules, parallel copies of the pipeline can be instantiated and the size of the bit-vectors can be increased. Using the Virtex 4, over 3x as many rules can be stored and a 6x improvement to throughput can be achieved, as shown in Figure 9. The figure shows the immediate improvements available for the number of rules, the frequency, and the throughput of the rule processor by targeting a newer FPGA.

5 Conclusions

FPGAs are well suited for implementing complex rule processing components, as needed for intrusion detection systems like Snort and intrusion prevention systems that protect the Internet. Reconfigurable hardware is efficient for header processing, string matching, and flow processing. To be an useful tool for network administrators, a framework was needed that puts these components together in an efficient way. Software-based implementations of complex rule processors have been shown to be too slow for processing data in high-speed networks. Hardware parallelism greatly improves performance. Through the use of FPGAs, rule processing systems can be deployed that provide the flexibility of adapting to the persistent changes required to be an effective network defense mechanism.

In this paper, a rule processing framework was presented that combines the three crucial components of network intrusion detection and prevention in a single system. The framework allows integration of modularized header and payload processing systems by adhering to well-structured interfaces.

References

[1] Vern Paxson, Stuart Staniford, and Nicholas Weaver, "How to 0wn the internet in your spare time," in *Proceedings of the 11th Usenix Security Symposium*, Aug. 2002.

[2] N. Weaver, D. Ellis, S. Staniford, and V Paxson, "Worms vs. perimeters: The case for hard-lans," in *Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2004.

[3] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland, "Characterizing the performance of network intrusion detection sensors," in *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Berlin–Heidelberg–New York, September 2003, Lecture Notes in Computer Science, Springer-Verlag.

[4] W. Lee, J. B. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang, "Performance adaptation in real-time intrusion detection systems," in *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Zurich, Switzerland, Oct. 2002, Lecture Notes in Computer Science, Springer-Verlag.

[5] John W. Lockwood, Christopher Neely, Christopher Zuver, James Moscola, Sarang Dharmapurikar, and David Lim, "An extensible, system-on-programmable-chip, content-aware Internet firewall," in *Field Programmable Logic and Applications (FPL)*, Lisbon, Portugal, Sept. 2003, p. 14B.

[6] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett, "Granidt: Towards gigabit rate network intrusion detection technology," in *12th Conference on Field Programmable*

Logic and Applications, Montpellier, France, 2002, pp. 404–413, Springer-Verlag.

[7] Martin Roesch, "SNORT - lightweight intrusion detection for networks," in *LISA '99: USENIX 13th Systems Administration Conference on System administration*, Seattle, Washington, Nov. 1999, pp. 229–238.

[8] David V. Schuehler, "Techniques for processing TCP/IP flow content in network switches at gigabit line rates," PhD dissertation, Washington University in St. Louis, 2004.

[9] David V. Schuehler and John W. Lockwood, "A modular system for FPGA-based TCP flow processing in high-speed networks," in *Field Programmable Logic and Application: 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004. Proceedings*, Antwerp, Belgium, Aug. 2004, pp. 301–310, Springer-Verlag.

[10] David Nguyen, Joseph Zambreno, and Gokhan Memik, "Flow monitoring in high-speed networks with 2D hash tables," in *Field Programmable Logic and Application: 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004. Proceedings*, Antwerp, Belgium, Aug. 2004, pp. 1093–1097, Springer-Verlag.

[11] Marc Necker, Didier Contis, and David Schimmel, "TCP-stream reassembly and state tracking in hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2002.

[12] Andy Currid, "TCP offload to the rescue," *Queue*, vol. 2, no. 3, pp. 58–65, 2004.

[13] R. Katz F. Yu, "Efficient multi-match packet classification and lookup with TCAM," in *12th Annual Proceedings of IEEE Hot Interconnects*, Stanford, CA, Aug. 2004, pp. 0–1.

[14] Haoyu Song and John Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *IEEE Internation Symposium on Field-Programmable Gate Arrays, (FPGA'05)*, Monterey, CA, Feb. 2005, pp. 238–245.

[15] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2002.

[16] Zachary K. Baker and Viktor K. Prasanna, "A methodology for synthesis of efficient intrusion detection systems on FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[17] Christopher R. Clark and David E. Schimmel, "Scalable multi-pattern matching on high-speed networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[18] James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos, "Implementation of a content-scanning module for an Internet firewall," in *FCCM*, Napa, CA, Apr. 2003.

[19] Y. Cho and W. Mangione-Smith, "Deep packet filter with dedicated logic and read only memories," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[20] Ioannis Sourdis and Dionisios Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004.

[21] Yutaka Sugawara, Mary Inaba, and Kei Hiraki, "Over 10gbps string matching mechanism for multi-stream packet scanning systems," in *Field Programmable Logic and Application: 14th International Conference, FPL 2004, Leuven, Belgium, August 30-September 1, 2004. Proceedings*, Antwerp, Belgium, Aug. 2004, pp. 484–493, Springer-Verlag.

[22] Michael E. Attig, Sarang Dharmapurikar, and John Lockwood, "Implementation results of Bloom filters for string matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, Napa, CA, Apr. 2004, pp. 322–323.

[23] Tak W. Yan and Hector Garcia-Molina, "Index structures for selective dissemination of information under the boolean model," *ACM Trans. Database Syst.*, vol. 19, no. 2, pp. 332–364, 1994.

[24] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, Monterey, CA, USA, Feb. 2001, pp. 87–93.