Specialized Hardware for Deep Network Packet Filtering

Young H. Cho, Shiva Navab, and William H. Mangione-Smith

The University of California, Los Angeles, CA 91311
{young, shiva_n, billms}@ee.ucla.edu
http://cares.icsl.ucla.edu

Abstract. Many computer network provide limited security through simple firewall feature in router and switch. Some networks that require higher security use deep packet filter to capture packets that can not be detected by simple firewall. Deep packet filters use list of rules for determining safety of packets. There is a high degree of parallelism in processing these rules because each rule represent independent pattern matching process. We find that the underlying architecture for existing software and hardware firewalls do not fully take advantage of this parallelism. Thus, we design a deep packet filtering firewall on a field programmable gate array (FPGA) to take advantage of the parallelism while retaining its programmability. Our implementation is capable of processing over 2.88 gigabits per second of network stream on an Altera EP20K series FPGA without manual optimization.

1 Introduction

The purpose of a Network Intrusion Detection System (NIDS) is to help protect computer network users from malicious attacks [1]. Today, firewalls with limited capabilities are built into many commercial network switch and router. Simple firewalls usually examine packet headers for specific information to determine whether to block or allow packet passage.

As computer networks grow faster, it becomes easier for the hackers to release malicious packets into a network without being detected. Simple firewalls may be able to protect networks from certain intrusion; but busy packet traffic and smart attacking schemes (e.g. Code Red) can easily by-pass such firewalls [2].

In order to identify such attacks, methods such as stateful multi-layer inspection are used in more advanced firewalls. A multi-layer inspection unit searches for a number of patterns in packet header. They furthermore perform computationally intensive pattern searches on the application level payload. A system with a multi-layer inspection unit offers a higher level of security than a traditional firewall. However, due to their complexity, such systems are slower and require better administration [3].

1.1 Motivation

There are several rule based multi-layer inspection firewall software packages available today. Most of these systems use one or more general purpose processors running rule-based packet filtering software. Due to exhaustive pattern detection algorithm, software system running on single general-purpose processor may not be able to inspect all network traffic [5–7].

On the other hand, there are some custom hardware chips (i.e. ClassiPi [9] and Strata II [10, 11]) that support faster network. However, the underlying sequential algorithm running on a Von Neuman architecture eventually leads to performance bottleneck as number of necessary pattern checks increase.

Therefore, we have designed a rule based multi-layer inspection firewall system based on a parallel architecture. We customize our hardware to process each rule separately in parallel.

Because network threats are constantly changing, it is very important to be able to change the rule set. In order to build a fast gate-level design that can be updated frequently, we choose FPGA as our design platform. In addition to speed and flexibility, FPGA tools allow quick extraction of realistic performance and resource costs.

1.2 Related Work

Research in reconfigurable network hardware assess the possibility of using reconfigurable hardware as a network processors [12], switches [13], routers [14, 15], and network protocol wrappers [16–19]. Projects such as PLATO of Technical University of Crete [13] and FPX of Washington University are reconfigurable network hardware design platform on which above application can be configured [14, 15, 17].

Reconfigurable packet filtering projects mentioned above are pattern matching units that make decision based on TCP and IP header information. Like most ASIC packet classification co-processors, the FPGA is configured to function as a co-processor to reduce the processing time for packet classification in few research projects [18, 19]. The reconfigurable nature of the FPGA adds additional flexibility in filtering mechanisms compared to ASIC solutions.

Firewalls which are limited to header processing can only partially satisfy the inspection requirements involved in deep packet search process. In deep packet search algorithm, many strings must be matched over application level payload as well as packet header. More thorough string pattern searches are required once the packet header analysis is complete. The performance cost in Von Neuman architecture of such pattern search is directly proportional to number of search pattern rules.

Sidhu and Prasanna mapped Non-deterministic Finite Automata (NFA) into FPGAs to perform fast pattern matching [20]. Using this method, Carver et. al compiled patterns for an open-source NIDS system into JHDL [21]. Due to parallel nature of the hardware design, the pattern matching engine maintain high performance regardless the size of the string.

1.3 Summary

Our implementation uses high-level rule description from the open-source firewall software Snort to build a complete stand alone NIDS. Therefore, the description of our NIDS begins with software architecture of Snort system in section 2. The algorithm and data structure of Snort becomes our high-level model for the hardware implementation. After understanding the software model, we describe the hardware implementation and its performance in section 3. Then we conclude in section 4 with a summary of our research.

2 Software Firewall

Several software based firewall units provide complex rule-based multi-layer inspection routine in addition to other security protocols. These firewall units can detect malicious packets that are not filtered out by simple header inspection [3]. In this section we describe the design structure of software stateful multi-layer firewall package.

2.1 Libcap based system

Libcap is a portable library of functions that can capture network packets and examine their length, contents, and header information. The result from the library function can be used to filter or classify network packets. Advanced software security application running on general purpose processor would use libcap or similar filtering library.

Snort is an open source lightweight network intrusion detection system that uses libcap [5]. Snort can perform traffic analysis and packet logging on IP networks, protocol analysis, and payload content searching. Furthermore Snort can be configured to detect a variety of abnormal packet behaviors, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, and OS fingerprinting attempts. The packet payload inspection is the key mechanism for detecting these malicious behaviors.

Hogwash is a software package that wraps around Snort to filter and forward packets between 2 networks [6]. This tool has the ability to generate alerts as well as packet drops and modification. Accordingly, the software can run on top of the network driver to stop attacks that cannot be blocked by a simple firewall. Although the rule format is slightly different between Hogwash and Snort, the underlying technology is the same.

2.2 Rule based system

Snort uses a list of rules to filter incoming packets. As the number of attacks grow, most commonly occurring attack patterns or dangerous patterns are turned into signatures which Snort parses. A simple rule structure and options allow flexibility and convenience in configuring the NIDS. However, as we

will describe in later section, there are performance disadvantages of having a long list of filter rules.

Snort uses a flexible rule language to describe traffic classification. It uses a detection engine that utilizes modular plug-in architecture. There are three primary subsystems that make up Snort: the packet decoder, the detection engine, and the logging and alerting subsystem. Snort maintain its detection rules in a two dimensional linked list of what are termed chain headers and chain options.

These rule chains are searched for each packet incident on the network. The detection engine checks only those chain options that have been set by the rule parser at run-time. The first rule that matches a decoded packet in the detection engine triggers the action specified in the rule definition and returns.

The structure of a rule consists of a command keyword for handling the matching packet, header information signature, and various options for other patterns including the content string search for the payload. Snort can use one or more rule files as its input. Each rule file can contain more than one rule signature consistent with the format shown as following.

```
Action Protocol SrcIPAddr/Port Direction DstIPAddr/Port Options
```

The first part of the rule corresponds to the packet header information. The second part corresponds to the pattern search options applied to the application level packet payload. The most computationally intensive option is called 'content.' This option is the key to better packet filtering in multi-layer inspection firewall.

Following rule is a signature that is used by Snort to detect one type of 'Code Red' worm.

```
alert TCP $EXTERNAL any -> $INTERNAL 80 (msg: "IDS552/web-iis_IIS
ISAPI Overflow ida"; dsize: >239; flags: A+; uricontent: ".ida?";
classtype: system-or-info-attempt; reference: arachnids,552;)
```

2.3 Experimentation

According to the documentation, a Hogwash system running on a celeron 733 with a moderate rule set can keep pace with a 100 Mbps network. We experimented with Hogwash and Snort on our computers to determine achieved performance in the field.

We ran Hogwash with 105 rules to filter the network traffic over a 100 Mbps fast Ethernet. With non-malicious traffic, the benchmark yielded a total throughput range of 39.2 to 58.82 Mbps. Then the test was repeated using malicious packets. As expected, throughput of the network was lower to range of 25.59 and 50.15 Mbps.

Accordingly, the results from our experiment suggested that Hogwash was not able maintain the full bandwidth on fast Ethernet. Although a software implementation is flexible and simple to implement, such a system running on single processor would not be able to accommodate network speeds above 100 Mbps.

3 Our Implementation

Hogwash is designed to block out about 95 percent of all the known Internet attacks using 105 signature rules of most common attacks. Thus, we decided to use these rule set to build our FPGA based NIDS.

The rules contain information to search through packet layers 2 through 4 first. If the packet header information matches the rule criteria, an exhaustive pattern search is performed on the payload (layers 5-7). Such a pattern search is done for all the rules with matching header information [5].

In our approach, each pattern matching component is automatically translated into structural VHDL. The components are then synthesized and mapped on to the FPGA with a vendor CAD tool set. Through the CAD tools, the design functionality is tested and the estimates for the resource requirement and performance are obtained.

3.1 Parallel Design

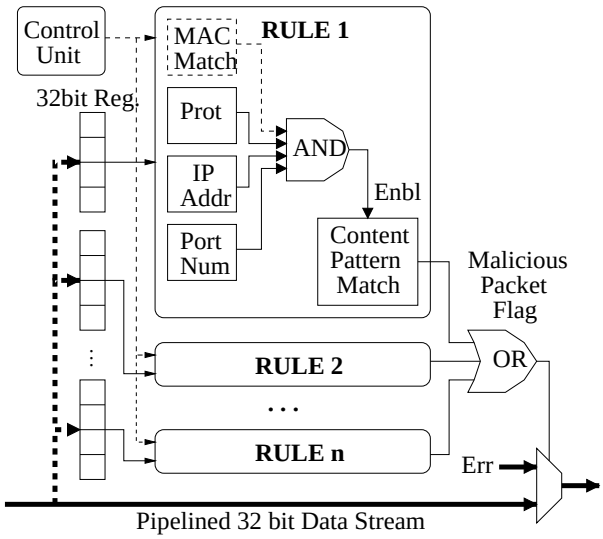

Fig. 1. Parallel Datapath of NIDS in Reconfigurable Hardware

Figure 1 is a block diagram of our architecture. Each rule unit implements the logic for a single Snort rule signature. Packet data is passed to the units through a 32-bit bus. Effective use of the hardware pipeline with optimized combinational logic between each stage shortened the critical path of the design.

The simplicity of the design easily shows its resemblance with the software execution path of Hogwash. The header information of each packet is compared

with the predefined header data. If the header information matches the rule, the payload is sent to the content pattern match unit where the predefine pattern is searched. However, unlike the software implementation, all the rule chains are matched in parallel to achieve predictable high performance.

In addition to parallel layout of the rule units, four bytes of data are matched in each stage of the pipeline to increase its throughput. Therefore, four instances of the search string are compared in parallel on every cycle to match through all the byte offsets of the 4-byte input stream.

If a given packet is detected to be malicious, a flag is raised. In a system with sufficient buffer memory, the flag can be used to drop the packet. However, without any external memory, this flag can be used to corrupt the rest of the packet payload going through the pipeline; thereby causing the packet to be dropped at the receiving node.

3.2 Content Pattern Match Unit

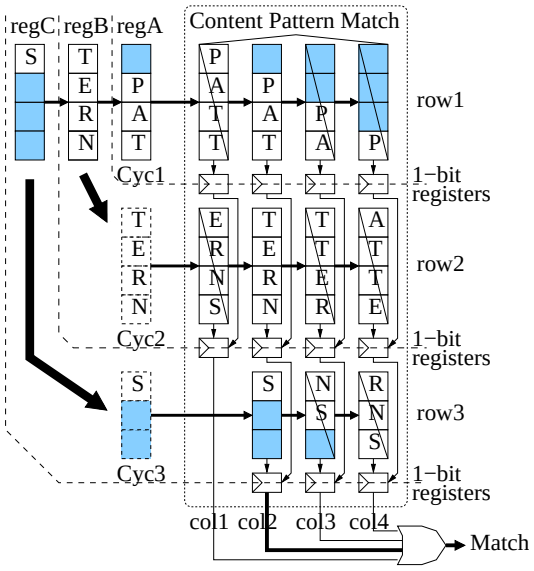

Fig. 2. Pattern Match Example

The content pattern match unit is the main work horse of the design. In order to clarify its functionality, we discuss an example of its operation.

We will first define the structure of the example diagram shown in figure 2. The squares outside of the content match unit are 8-bit register. The solid lined square shows their initial state while the dotted lined squares are the subsequent future states of the registers. The shaded squares represent don't care values while the rest of the squares contain ASCII characters.

The squares within content match unit represent 8-bit comparators. The four 8-bit comparators work together to match four consecutive bytes of data simultaneously. The match results are passed to the 1-bit registers located below all the comparators. Output from 1-bit register control the subsequent 1-bit comparison result.

The content of the register A, B, and C represent the string stream 'PATTERNS.' Since the registers are pipelined, we can separately observe different parts of the design during each clock period. The matching stages are indicated as clock cycles 1, 2, and 3.

During cycle 1, substring 'PAT' is compared against four different strings with different byte offsets in row 1. Since all of the 1-bit registers are initialized as 0, the result will not be latched even if there is a matching pattern in rows 2 and 3. For our example, the comparator in row 1, column 2, produces 1. This value is latched through the 1-bit register to enable the next row of 1-bit registers.

In cycle 2, the substring in register B is latched into register A. The substring is then matched with all the patterns. However, only comparator results that can be latched are all the comparators in row 1 and the comparator in row 2, column 2. The purpose behind enabling all the row 1 registers is because of a possibility that the pattern may have actually started with the content in register B during cycle 1. The matching pattern in row 2, column 2, enables the 1-bit register in row 3, column 2 for the cycle 3.

Finally, with the content of register C in register A in cycle 3, substring 'S' is compared with comparators in row 3. The match in column 2 comparator sends 1 to the OR gate which indicate the match of the substring 'PATTERNS.' This signal then can be used to alert the user or drop the packet.

All contents for the rules are compared in parallel. Although such construction suggests growth of design area linearly proportional to the number of the rules, the performance of the filter is maintained as a constant.

Since all the comparators compare the incoming stream of data with fixed value, it is unnecessary to keep the pattern values in a register. Rather, the effect of comparator can be achieved by using 32-bit AND gate with inversion in the input lines corresponding the zero bits of the bit pattern. Therefore the hardware requirement is much smaller than implementing comparators.

3.3 Methodology

In a production environment, we need to continually update the rule set to guard against new attacks. Thus, flexibility of updating the rules is valuable, if not essential. Consequently, we choose FPGAs as our development platform. Efficient pipeline structure and simplicity of our design algorithm also allows FPGA to perform well.

In order to efficiently update the design upon changes in the rule set, we automate the design process. We extract templates from the optimized structural VHDL files. Then the Hogwash rule file is pre-processed and converted to VHDL files based on the template.

The resulting structural VHDL files are highly pipelined and optimized with user-defined parameter because of the efficient design templates. These VHDL designs can be inserted into the common datapath to complete the system.

FPGA code is produced with the net list of the design. The net list is processed and mapped on to a specified FPGA with a place and route tool. In most cases, the larger match rules cause slower place and route and low performance design. However, due to highly pipelined and parallel architecture, there is more freedom for place and route tool to produce efficient implementation in shorter amount of time.

3.4 Performance

In a large FPGA design, in addition to correct functionality, the critical path and the resource usage information are main concerns. These results are only available at the end of place and route step of the FPGA design process.

Many commercial place and route programs support optimization options. For this paper, we do not use these options. Thus, all our results are based on an FPGA implementation without any timing or resource optimization.

Our implementation use Quartus II FPGA tool from Altera to obtain the final design as well as timing and resource information for single rule implementation of Hogwash.

Our NIDS design is mapped on to the Altera EP20K series FPGA chips. The basic construction units are called "logic elements" (LE) ranging from hundreds of LE on smaller devices up to fifty-two thousand LE on the largest of the series. There are also larger and faster FPGA series available but we found this series more than suitable [22].

The design is placed and routed without invoking any optional timing restriction allowed in the tool. Therefore, the critical paths are not optimized. With the longest clock to signal delay of 11.685 ns for reset signal, the system can run at 90 MHz. At 90 MHz, the system can filter 2.88 Gbps of data regardless of the size of the patterns or the packet length. This is in contrast to the work by Craver et. al which achieved approximately 6.4 Mbps (800 Kilobytes/sec).

With some timing optimizations, the implementation should be able to run at a faster clock rate. However, such optimization would require longer time for place and routing process. Although timing restrictions can take longer place and route time, it maybe necessary to achieve higher throughput. Design time for new reconfiguration may be acceptable in most cases; this is because of the infrequent updates to the Hogwash rule set.

3.5 Resource Requirement

Resource requirement for content pattern matching units vary according to the length of each string. The resource usage report from the Altera place and route tool indicate that up to 10 logic elements are required to implement 1-byte comparator on EP20K series chip. This resource requirement is proportional to the total number of characters defined in the content option of all the rules.

Most of the current rules for Hogwash does not examine IP addresses and source port number. Therefore, the logic resources required for the header comparator, with exception of destination port inspection for each rule, is negligible.

Since a fixed number of logic elements are required for the control and datapath (approximately 250 LE) Hogwash rules requiring a total of 1611 byte comparison from 105 rules may be implemented with roughly 17,000 logic elements. Therefore, we estimate that the current Hogwash rule set can be mapped on a mid-size Altera EP20K FPGA with 20,000 LE.

3.6 Limitations

For the sake of simplicity, some features of Snort were not implemented in our design. One simple option would be case insensitivity in string comparisons. One way to implement case sensitivity in hardware is by checking the range of the characters and simply subtracting the case offset before sending the data to the content pattern match unit. This will require an additional twelve 8-bit comparators for each rule with the nocase option. In such manner, this limitation can be addressed.

A more difficult limitation to overcome is handling packet fragmentation. To overcome this limitation, the implementation requires buffers to store all the packets in a stream until they can be reassembled. Defragmentation is currently addressed in software.

4 Conclusion

The benchmark details of current multi-layer inspection packages reveal performance and functional shortcomings. The performance slow down is due to the sequential execution of a large set of pattern matching rules on each packets. Therefore, existing software or hardware may not be fast enough to filter out the complex malicious attack for the networks with bandwidth above 100 Mbps.

Our parallel implementation can filter network traffic at 2.88 Gbps. By examining the critical path information, we hypothesize that our implementation can filter network traffic with bandwidth of 6 Gbps by turning on some timing constraints on the place and router of the FPGA.

Given sufficient amount of hardware resource, all the filter rules can process the packet in parallel. A highly pipelined design gives fixed high performance for even the most complex pattern search rule. Accordingly, our experimental results indicate full filtering capability on a data bandwidth of several gigabits per second.

References

1. J. McHugh, A. Christie, J. Allen, "Defending Yourself: The Role of Intrusion Detection Systems," IEEE Software Magazine, Sept./Oct. 2000.

2. CERT/CC, "CERT Advisory CA-2001-19 Code Red Worm Exploiting Buffer Overflow In IIS Indexing Service DLL", Carnegie Mellon Software Engineering Institute, August 23, 2001.
3. Vicomsoft Inc., "Firewall Q&A", 2001.
4. J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, E. Stoner, "State of the Practice of Intrusion Detection Technologies," Technical Report, Carnegie Mellon Software Engineering Institute, Jan. 2000.
5. M. Roesch, "Snort - Lightweight Intrusion Detection for Networks", USENIX LISA '99 conference, Nov. 1999.
6. M. Karagiannis, "How to create a Stealth Packet Scrubber using Hogwash," Application Notes for Hogwash, 2001.
7. Netperf.org, "Netperf documentation", 1997.
8. SonicWall Inc., "Denial of Service Attacks - An Emerging Vulnerability of "Connected" Network," White paper, 2001.
9. Broadcom Inc., "Strada Switch II BCM5616 - Integrated Multi-layer Switch," Product Brief, 2001.
10. PMC Sierra Inc., "PM2329 ClassiPi Network Classification Processor Datasheet," Product Datasheet, PMC-2010146, Issue 4, 2001.
11. PMC Sierra Inc., "Preliminary PM2329 ClassiPi Wire-speed Performance Application Note," Application Note, PMC-2010258, Issue 1, October 2001.
12. M. Iliopoulos, T. Antonakopoulos, "Reconfigurable network processors based on field programmable system level integrated circuits," 10th Conference on Field Programmable Logic and Applications, pp. 39-47, 2000.
13. A. Dollas, D. Pnevmatikatos, N. Aslanides, et al., "Rapid prototyping of a reusable 4x4 active ATM switch core with the PCI pamette," 12th International Workshop on Rapid Prototyping, pp. 17-23, 2001.
14. J.W. Lockwood, "Evolvable Internet hardware platforms," Proceedings of the 3rd NASA/DoD Workshop on Evolvable Hardware, pp. 271-279, 2001.
15. F. Braun, J. Lockwood, M. Waldvogel, "Reconfigurable router modules using network protocol wrappers," 11th Conference on Field Programmable Logic and Applications (FPL01), pp. 254-263, 2001.
16. H. Fallside, M.J.S. Smith, "Internet connected FPL," 10th Conference on Field Programmable Logic and Applications, pp. 48-57, 2000.
17. F. Braun, J. Lockwood, M. Waldvogel, "Protocol wrappers for layered network packet processing in reconfigurable hardware," IEEE Micro, Vol. 22, Issue 1, pp. 66-74, Jan.-Feb. 2002.
18. P.W. Dowd, J.T. McHenry, F.A. Pellegrino, T.M. Carrozzi and W.B. Cocks, "An FPGA-Based Coprocessor for ATM Firewalls," Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines (FCCM97), April 1997.
19. R. Sinnappan and S. Hazelhurst, "A Reconfigurable Approach to Packet Filtering," In Proceedings of FPL 2001: 11th International Conference on Field Programmable Logic and Applications, Belfast, United Kingdom, August 2001.
20. R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01), April 2001.
21. D. Carver, R. Franklin, B.L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines (FCCM02), April 2002.
22. Altera Inc., "Altera Quartus II Development Software Manual", 2001.