Int. Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan.

Rule Set Decomposition for Hardware Network Intrusion Detection

Timothy Ramirez and Chia-Tien Dan Lo
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas 78249
{tramirez, danlo}@cs.utsa.edu

November 5, 2004

Abstract

This paper describes the work being done to minimize hardware requirements for a hardware-assisted Network Intrusion Detection System (NIDS). This system will use a core Intrusion Detection System (IDS) in a distributed manner. The distribution is beneficial for two reasons. First, the system will not have a single point at which an attacker can direct attack traffic for the purpose of overloading the IDS. Second, due to the number of attacks and corresponding signatures it is not feasible to design a system using programmable hardware that recognizes all signatures. The NIDS *Snort* will be augmented by passing the packet matching function to a Field Programmable Gate Array (FPGA). *Snort's* rule set consists of tens of hundreds of "signatures" and is decomposed to minimize the capacity of FPGAs necessary to implement the entire rule set. The circuit is based on a Finite Automaton where each character represents a state. First, the rules are broken down to common groups that share similar characters. These groups are then used to decompose the entire rule set into logical sets whose patterns can be matched with a simple string matching circuit. After reducing the rule set by taking advantage of character repetition we are left with about 51% of the states necessary to match all of the patterns. This state reduction translates to a smaller circuit used to match all of the patterns and the circuit can be implemented in as few devices as possible. [1]

1 Introduction

According to reports from the Computer Emergency Response Team (CERT) network attacks [6], specifically Denial-of-Service (DoS) attacks, have increased dramatically over the past few years. In part, the increase is due to the popularity of the Internet and the wide-spread vulnerabilities of different systems operating on the Internet. In an attempt to limit these attacks, administrators employ different security tools that can stop the attacks, or at the very least notify them that an attack has occurred or is occurring. Network Intrusion Detection Systems (NIDSs) are used to detect the existence of attack traffic by continually monitoring the network's traffic. The NIDS looks for specific behaviors, packet formats or packet contents. When a match, or a deviation from some normal behavior, occurs the system takes an appropriate action. This action can take the form of logging the event or, ideally, stopping the malicious traffic.

Signatures for attacks are developed by analyzing the traffic for a particular attack and selecting packet parameters that are representative of the traffic, i.e. source address, packet type, packet contents. When examing a packet the content of the packet may be of significant importance in determining its legitamacy. The NIDS *Snort* examines a packet's content based on its headers, which indicates the packet is suspicious. The algorithms for pattern searching can be very time consuming and can take up to 60% of a packet's processing time [4]. Therefore, more efficient methods must be employed to search a packet's payload for a specific pattern. Once a

[1]Partial support for this work is provided by the Center for Infrastructure Assurance and Security at UTSA through grants CIAS-26- 0200-6275.

pattern has been found, or not found, the packet can be classified as malicious or benign.

As network speeds increase, the NIDS must be capable of examing network traffic at network line speeds. *Snort* claims to be capable of operating at a network speed of 1 Gbps [5] although in practice reliable operation speeds are normally less than a few hundred Mbps. Networks operating at 10 Gbps are becoming more common; hence, NIDSs will need to operate at this speed to be effective in examing all network traffic. One solution is to move the computationally intensive task of pattern matching to hardware that is dedicated to matching the known patterns. The hardware can then be updated periodically to include new patterns as new signatures are learned. This paper focuses on reducing the size of the pattern matching logic in order to minimize the hardware requirements.

2 Background and Related Work

2.1 Regular Expressions and Finite Automata

In order to identify a sequence of characters, or a pattern, the pattern must be described by some method to represent the sequence of characters. Regular expressions have been developed to describe character sequences. A regular expression uses characters and meta-characters to describe not only character sequences but also pattern behaviors such as repeating characters and alternation among different characters. Figure 1 is an example of a regular expression that matches the character sequences “this” or “that”. This regular expression can then be used with a utility like *grep* to find the sequence of characters that match the expression within a larger sequence of characters. This method can be employed to parse a stream of characters for a chosen pattern. The *grep* utility constructs, to some degree, a Finite Automaton to determine if a match is present in the larger sequence of characters.

A Finite Automaton (FA) is a collection of states which are connected by directed edges representing transitions between two or more states. The edges of an FA are marked by the character, or characters, that cause a

PATTERN	REGULAR EXPRESSION
"this" or "that"	th(is|at)

Figure 1: Regular expression for “this” or “that”

transition. Figure 2 is the FA that represents the previous regular expression in Figure 1. There are two types of FAs; non-deterministic FAs (NFAs) and deterministic FAs (DFAs). NFAs can be distinguished from DFAs by the fact that there may be more than a single transition with the same input or there may be empty transitions. Because of these differences, NFAs can be inefficient due to the overhead involved with keeping track of multiple possible states with a state transition table. A DFA can be more effiecient than an NFA since only a single transition is possible given a single input. The drawback to DFAs is that it can be expensive in terms of space and construction time. The DFA’s strength lies in the time required to determine a match, which is linear in terms of the searched text’s length. The worst case time to determine a match for NFAs, by comparison, is quadratic.

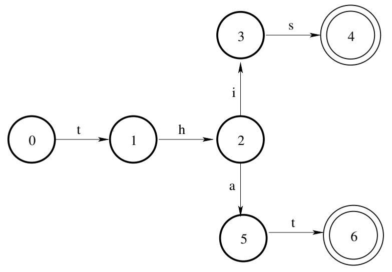

Figure 2: Finite automaton for matching “this” or “that”

2.2 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are devices that can be configured to implement task specific logic. They are a middle ground between software and hardware that allows for the flexibility of software while gaining the speed advantage of hardware. In order to configure an FPGA, the logic must be described using a hardware description language like VHSIC Hardware Description Language (VHDL). Once described, the VHDL is compiled and the resulting configuration bits can be used to "burn" the configuration onto the device. The FPGA can be re-configured later if a new logic becomes necessary.

2.3 Snort

Snort is a popular open source network-based IDS. Functionality is very diverse, especially with the inclusion of preprocessor modules. According to the documentation, *Snort* is capable of handling traffic up to around 1 Gbps [5]. Our tests result in a monitored bandwidth of less than 60Mbps while other tests show *Snort's* limitation to be between 100 and 200 Mbps with reasonably good performance in terms of intrusion detection, i.e. no dropped packets. The preprocessor *perfmonitor* is used to get statistics from *Snort's* operation. This includes the speed at which packets are examined and the amount of time involved in performing pattern matching. These two key metrics are of particular interest to this research since we are attempting to improve the speed of *Snort* through improved pattern matching. Due to the amount of pattern matching performed by *Snort*, it is clear that speeding up the pattern matching process will benefit *Snort's* overall speed.

2.4 Related Work

The methods we employ to implement the pattern matching logic have been researched previously. A paper by Sidhu and Prasanna explains how to implement regular expression matching in hardware [7]. They break the problem down into two components; regular expression generation and logic configuration. By using well known definitions of FAs, Sidhu and Prasanna explain the different logic structures and how they can be mapped together to derive a match for a given pattern. The focus of their paper is to show the improvement over a software approach to pattern matching that hardware can provide. A paper by Sourdis also uses FPGAs to improve pattern matching, although their implementation uses a distinctly different logic with comparators and shift registers [8]. Sourdis' work implements the pattern matching for content searching of network packets that is directly applicable to our work. Sidhu and Prasanna only apply their techniques to the general problem of pattern matching. Another paper by Franklin and Hutchings [3] implements pattern matching for *Snort* using Sidhu and Prasanna's approach. This work is a couple of years old and *Snort's* rule set has grown to a size that may not fit on a single device. All of the mentioned approaches are limited by the fact that an FPGA has a fixed number of logic gates available for configuration. This results in not being able to fully implement a large rule set. Our proposal helps address this limitation through rule set decomposition.

3 Rule Set Decomposition

Our proposal will use *Snort* as a foundation NIDS. It will be augmented to increase performance to handle high speed traffic without dropping packets. We can accomplish this by using FPGAs to implement a packet matching module. We attempt to reduce the size of *Snort's* rule set to overcome the problem of device limitations. *Snort* uses rule files that describe packet parameters that represent known attacks. There are numerous rules that are grouped together by the type of attack. The different rules can be included in *Snort's* rule set by specifying the relevant rule files during configuration. Previous implementations of pattern matching for *Snort* rules is summarized in Table 1 [8]. As can be seen from the table, the creation of a circuit that implements pattern matching for approximately 210 patterns, or 2500 characters, uses about 71% of device capacity. There are over two thousand rules that come with *Snort v2.1.1.* Due to the limited space on an FPGA, it will be necessary to use multiple devices for the entire rule set. Therefore, the goal is to reduce the rule set and implement them on as few devices as possible.

This can be accomplished by generating a regular expression for all of the patterns and constructing an NFA to determine if there is a match. The naive approach would be to generate a long regular expression that would match all patterns and construct the NFA for that expression. Due to cost considerations, the naive approach would result in an inefficient implementation of the entire rule set. To minimize the number of necessary states in the NFA we decompose the rules based on

Device	Util.	Num of patt.	Chars per patt.
Virtex 1000	7%	10	10
	33%	47	10.4
Virtex2 1000	16%	10	10
	80%	47	10.4
Virtex2 6000	71%	210	11.7

Table 1: Sourdis' results for FPGA utilization

header information and content. Additionally, by grouping rules with similar content together we can reuse the pattern matching logic, thereby reducing the total number of states necessary to implement all of the rules. For example, if two rules look for the content "this" and "that" then the logic that identifies "th" can be reused. The naive approach would use eight states for the eight characters t, h, i, s, t, h, a and t. If we reuse the logic for 't' and 'h' then the total number of states necessary would be six; t, h, i, s, a, t. Figure 2 shows the FA for the second approach.

3.1 Pattern Matching Logic

In order to match specified patterns, the hardware must be configured as an FA for the regular expression obtained by decomposition. We can start with the graph representations of the FAs as described in [1]. The FAs for a "or" b and a "and" b are shown in Figure 3. Transforming the graphs into hardware logic components can be done using the methods presented by Sidhu and Prasanna [7]. Their method builds the basic components necessary for implementing the four constructs of a pattern macther; a character matcher, the alternation operator, the catenation operator and the Kleene star (*). Our proposal can not use the Kleene star operator since payload content is of a specific size and character occurrences in attack signatures are bounded. Figure 4 shows a mid-level schematic for both the 'and' and 'or' logic. The structures S0 and S1 are composed of a flip-flop and character matcher as illustrated in Figure 5. The flip-flop holds the value of the state during a given clock cycle as detertmined by the enable signal. The character matcher uses two four-bit Look-Up Tables (LUTs) to match a byte from the bus. The next step

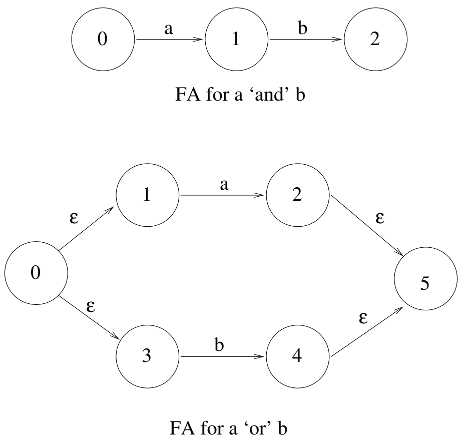

Figure 3: Finite automaton for 'or' and 'and' operations

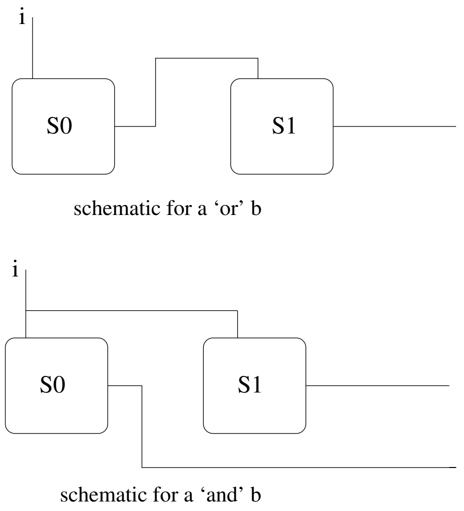

Figure 4: Mid-level schematic for 'or' and 'and' circuit

is to implement the derived regular expressions on the device by using combinations of the basic logic structures. The logic can be combined by connecting the appropriate outputs to the appropriate inputs to form a complete circuit. Figure 6 shows the circuit for matching the pattern "abc". Each of the states are represented as flip-flops with the signal propagating through the circuit only when there is a character match and the state accepting the character is active. The final design will have a circuit that implements the least number of states possible for all of the patterns in the rule set. While an NFA is constructed to match all of the states it is im-

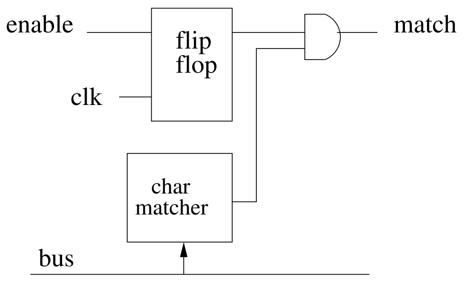

Figure 5: State logic components

portant to notice that the running time to find a match is not quadratic with the size of the searched text as the software implementation would be. The running time is linear with the size of the searched text. This is possible because of the nature of hardware and the parallelized matching that occurs with signal propagation.

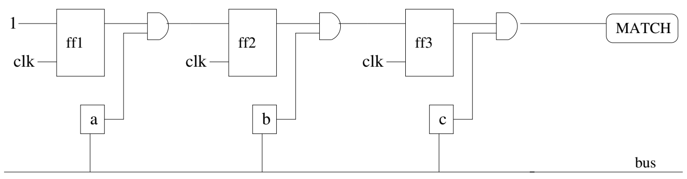

Figure 6: Schematic for 'abc' matching circuit

4 Decomposition Results

Using the naive approach and building a regular expression based on all of the extracted patterns, the FA would have close to 30,000 states representing the number of characters in all of the patterns for all of the rules. A simple reduction can be made by implementing a pattern only once. There are close to one thousand patterns that are common with a previous pattern in the rule set. By reusing the logic for these redundant patterns we get a 21.3% reduction in the number of states simply by requiring the patterns be unique. Further reductions can be made by examing different patterns but looking for similarities in those patterns. For example, if we group the unique patterns by common first character and reuse that initial state of the group for each of the group members we can get a further reduction of 7.8%. Taking the last approach and maximizing the similarities by looking for the longest common prefix we can reuse the maximum number of states in our FA. This achieves a reduction of close to half, 51.0%, of the original number of states necessary for all of the patterns. Table 2 summarizes our results.

Method of reduction	Original num of states	Reduced states	Percent reduced
Naive (no reduction)	29403	29403	0.0%
Unique patterns	29403	6270	21.3%
Common first char	23133	1813	7.8%
Longest prefix	23133	8148	35.2%
Total reduction	29403	14418	49.0%

Table 2: Decomposition results

5 Conclusion and Future Work

Implementing some form of decomposition on a large rule set is necessary to reduce the size of hardware requirements. By grouping rules together into logical subsets we are able to maximize the use of an FPGA. This is achieved by reusing the logic for similar characters. We have presented that we can reduce the number of states necessary to implement *Snort's* rule set to about 51.0% of the original number of states. While our reduction is promising we still must implement header matching logic to correlate the patterns to specific packets in order to reduce the number of false positives that would result in simply matching patterns. This additional circuitry will undoubtably increase the capacity beyond a single device. Our work will continue in this direction and organize a distributed system to split the workload among the multiple devices that will be necessary when all rules have been implemented.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Brian Caswell, Jay Beale, James C. Foster, and Jeremy Faircloth. *Snort 2.0 Intrusion Detection*. Syngress, 2003.

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

[4] Chia-Tien Dan Lo. "Hardware-Assisted Network-Based Intrusion Detection". In *International Conference on Informatics, Cybernetics, and Systems*, Kaohsiung, Taiwan, December 14-16 2003.

[5] Martin Roesch. *Snort User Manual, 2.1.1*. http://www.snort.org, February 25 2004.

[6] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. "Practical Network Support for IP Traceback". In *ACM Special Interest Group on Data Communication, SIGCOMM'00*, pages 295–306, Stockholm, Sweden, 2000.

[7] Reetinder Sidhu and Viktor K. Prasanna. "Fast Regular Expression Matching Using FPGAs". In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Rohnert Park, CA, USA, 2001.

[8] Ioannis Sourdis and Dionisios Pnevmatikatos. "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System".